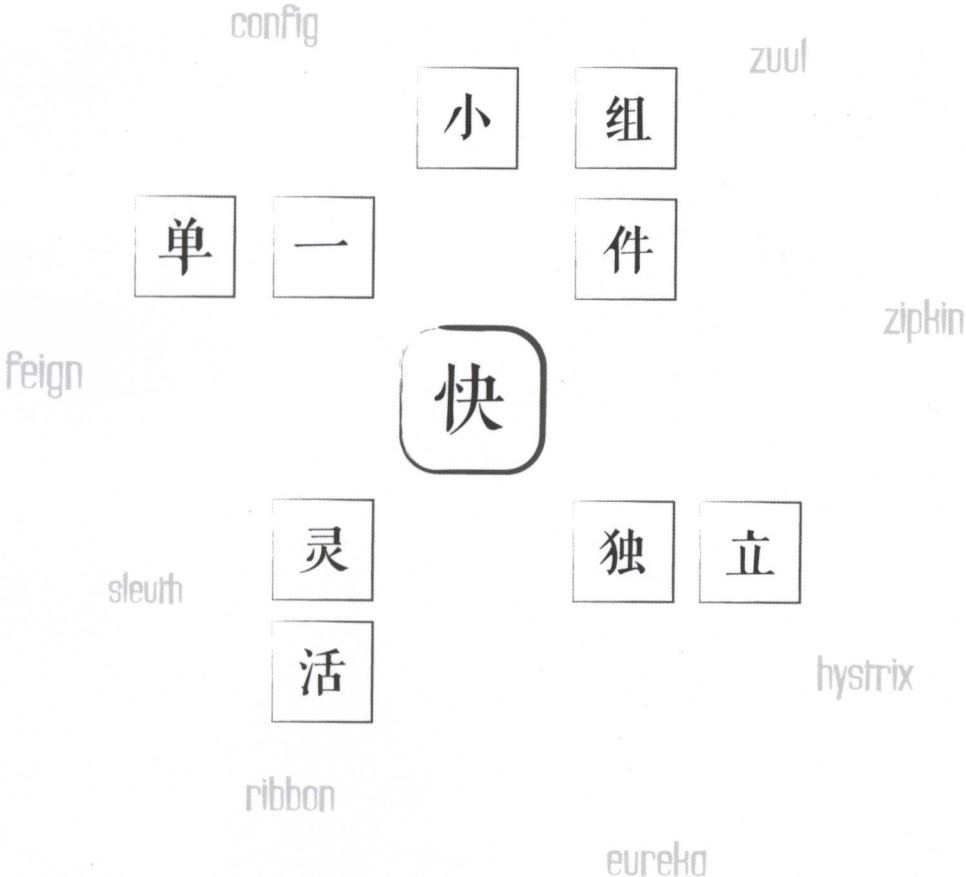
版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

解密微服务 Spring Boot Spring Cloud 技术实践

微服务

那些事儿

纪晓峰 ◎ 著



中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn



纪晓峰

现任某公司架构师，曾经在AC担任微服务领域的咨询师，长期供职于HP，具有丰富的架构设计和开发经验，先后参与多个微服务项目。

网名“叫我哲学家”，喜欢看书，钻研技术，领悟其中原理，习惯于独立思考，形成独到的见解。

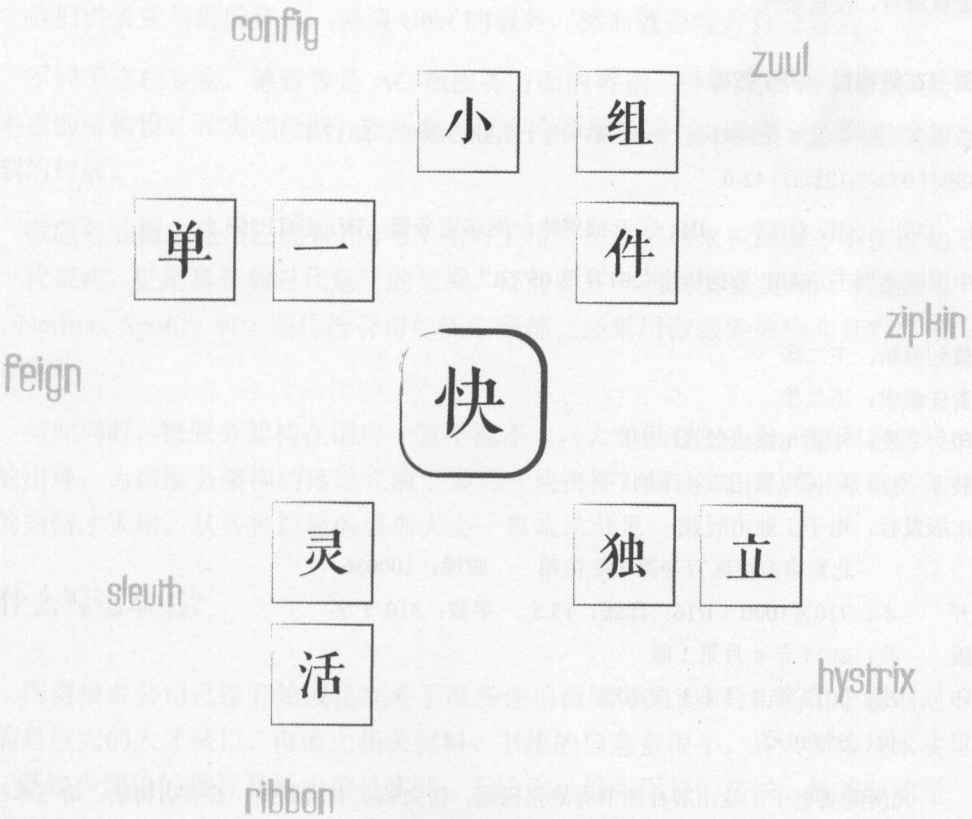
联系方式：171928202@qq.com

解密微服务 Spring Boot Spring Cloud 技术实践

微服务

那些事儿

纪晓峰 ◎ 著



電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

本书以幽默风趣的语言生动地讲述了微服务，将微服务相关的理论知识、技术实现和项目实践一网打尽。全书共分为三大部分：第一部分为“微服务解惑篇”，让读者对微服务有一个清晰的认识，消除对微服务的种种疑惑，同时对用户十分关注的如何拆分、如何使用等问题做了很全面的讲解；第二部分为“技术实现篇”，通过对当前世界最为流行的微服务框架 Spring Boot+Spring Cloud 的常用组件的详细解读以及其他相关技术、工具的介绍，来帮助读者了解整体架构及其中各个组件的原理，结合代码帮助读者快速构建项目，搭建一个比较完整的微服务框架；第三部分为“项目实战篇”，通过案例来展示不同业务系统在微服务使用过程中如何梳理业务、如何拆分、如何使用微服务重构等。

本书不仅适合架构师、开发人员、测试人员及运维人员阅读，也适合正在尝试使用微服务架构解耦历史遗留系统的团队或个人参考，希望本书能在实际工作中对读者有所帮助。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目 (CIP) 数据

微服务那些事儿 / 纪晓峰著. —北京：电子工业出版社，2017.4

ISBN 978-7-121-31142-0

I. ①微… II. ①纪… III. ①互联网络－网络服务器 IV. ①TP368.5

中国版本图书馆 CIP 数据核字(2017)第 057517 号

策划编辑：王二华

责任编辑：王二华

印　　刷：中国电影出版社印刷厂

装　　订：中国电影出版社印刷厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱　　邮编：100036

开　　本：720×1000　1/16　印张：13.5　字数：310 千字

版　　次：2017 年 4 月第 1 版

印　　次：2017 年 4 月第 1 次印刷

定　　价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010)88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：(010)88254532。

前　言

当莱斯特城获得 15/16 赛季英超冠军的时刻，我毫不犹豫地亮出了自己隐藏了 30 年的狐狸城球迷身份。

随着微服务在国内的炙手可热，很多人亮出了隐藏多年的微服务专家身份。

然而有一种专家就是那种告诉你 1000 种谈恋爱的办法，但自己从来没有谈过恋爱的人。

他们讲讲亚马逊的订单，谈谈 Uber 的服务，然而就是没有自己做的。

不同于这些专家，笔者曾是 AC 微服务方面的咨询，曾长期供职于 HP，具有丰富的架构设计和实战经验，先后参与过多个微服务项目，对微服务有着较为深刻的理解。

微服务的概念提出已经有几年了，相对于传统技术架构来说微服务不仅仅是新一代架构，更是具有划时代意义的架构。目前国外一些大型互联网公司如亚马逊、Netflix、Spotify 和一些传统公司如沃尔玛都已经采用微服务架构并且实际效果非常好。

与此同时，微服务架构在国内一直不温不火，大多呈观望态势，直到容器技术的出现，为微服务架构的落地完成了最后一块拼图，国内 IT 公司对微服务架构的热情才大增，从各种微服务培训大会一票难求可见一斑。

为什么写这本书?

国内很多公司已经开始或正在着手准备使用微服务架构，与市场需求暴增对应的是巨大的人才缺口，市面上相关材料、书籍的信息量很小，且大多偏技术实现，既缺少理论的指导又缺少项目实践，无法满足最终用户的需求。最重要的是还没有介绍 Spring Boot+Spring Cloud 的书问世，它可是当前世界上最主流的微服务技术实现，这多少影响了微服务在国内的发展。

技术类书籍大多有一个普遍的问题，就是枯燥乏味、晦涩难懂。不要说非技

术人员看不懂，连技术出身的人看着也吃力，更别提那些翻译过来的技术书籍。

究其原因是因为描述技术时用的语言是技术语言，而不是生活语言，自然难以理解。

技术不是玄学，技术应该是面向生活的。

所以本书会通过大量生活中的例子对微服务架构及原理进行解释，有趣易懂，可以说是接地气的一本技术书籍。

本书内容都是从实际出发，均来自本人常年在前方发回的“报道”，这里有客户最关心的话题，如何拆分服务、如何使用微服务等。本书有针对性地对这些问题进行了解答，同时提供了一套完整的理论体系，让读者知其然知其所以然。

一本书是一个作者智慧的结晶、知识的沉淀、思维体系的形成。

做人呢，开心最重要。

写书呢，态度最重要。

本人能力一般、水平有限，如有不足之处，欢迎“拍砖”。如果通过这本书能让更多人了解和更好地使用微服务，那么我的目的就达到了。

您适合看这本书么?

本书的最大特点是贴近生活，简单易懂，同时又与实践紧密联系。这本书的读者不仅限于技术人员，同时也适用于非技术人员。技术人员看完可以轻松地搭建一套完整的微服务环境，开始微服务之旅；架构师看完可以规划设计出更好的架构，业务人员看完也可以从中了解微服务。

只要是对微服务有兴趣的同道中人，都可以从这本书中有所收获。就像Spring Cloud的工具包，内容丰富，如果要使用微服务总有一款适合你。

本书是如何组织的?

本书分为三大部分。

第一部分微服务解惑篇包含了前5章

微服务会给很多人带来疑惑，它是什么，可能一说就懂，但是该不该使用它，以及如何使用等问题就不是三言两语能讲得清楚、说得明白的。这部分内容可以

让读者对微服务有个清晰的认识，消除对微服务的种种疑惑，同时对用户十分关注的如何拆分、如何使用等问题做了很全面的讲解。可以让之前的迷惑与误解消失不见。

第 1 章：微服务架构

本章介绍了微服务架构的产生背景、特点、优势及不足，还有在使用过程中会遇到的一些挑战，让读者对微服务有一个整体的了解。

第 2 章：为何选择微服务

本章主要讲选择微服务的理由，介绍传统架构的缺陷，而微服务就是它的“解药”，随后介绍了微服务带来的价值及通过它最终要达成的目标，最后介绍了成功人士的选择，让你用着放心。

第 3 章：我拆我拆我拆拆拆

本章介绍了微服务的拆分原则，以及拆分过程中需要注意的地方。它是微服务使用过程中非常重要的一环。

第 4 章：如何使用微服务

本章介绍了微服务在使用时应该如何规划，并以秦国都城举例说明规划的重要性及规划的方法。随后介绍微服务的使用方法，让读者在使用过程中有一个理论上的指导。

第 5 章：微服务的朋友圈

本章介绍了微服务架构的圈子，它不是一个独立的存在，有一系列热门词围绕着它，如容器、DevOps 等。

第二部分技术实现篇包含了第 6～9 章

通过对当前世界最为流行的微服务框架 Spring Boot+Spring Cloud 的常用组件的详细解读以及其他相关技术、工具的介绍，来帮助读者了解整体架构及其中各个组件的原理，结合代码帮助读者快速构建项目，搭建一个比较完整的微服务框架。另外，对一些误读进行了澄清，如 Spring Boot 究竟是不是微服务框架。

第 6 章：Spring Boot

本章介绍了基础框架 Spring Boot 的一些情况，包括特点、如何使用等。

第 7 章：Spring Cloud

本章介绍了 Spring Cloud 中比较常用和重要的一些组件的原理和使用，如

Eureka、Config、Zuul、Hystrix、Ribbon 等。

第 8 章：其他相关技术和工具

本章介绍了微服务的使用过程中需要用到的一些工具和技术，如 Liquibase、Swagger、Kafka、Spring Security 等，这些工具、技术可以辅助微服务被更好地使用。

第 9 章：测试相关

本章介绍了微服务在使用过程中测试的重要性，以及需要用到的一些测试方法和工具，如 Mockito 等。

第三部分项目实战篇包含了第 10～11 章

第 10 章：三个典型系统案例

本章通过三个案例来展示不同业务系统在微服务使用过程中如何梳理业务、如何拆分、如何使用微服务重构，等等。

第 11 章：开发管理

讲了开发管理中的管理原则、日常管理、代码质量管理、工作方式及相关人员的工作职责等内容。

我要致谢的人

首先感谢我的妻子，妻贤夫安，谢谢妻子对我的理解和包容，给我创造了愉悦的写作环境。谢谢妻子对家的付出，让我无须操心家中事，将更多的精力投入到写作中。谢谢妻子在我写作遇到瓶颈时给予的支持和鼓励，感恩有你的陪伴。

我还要感谢我曾经效力多年的中国惠普，一家非常伟大的公司，惠普之道让我有家一般的归属感，我在这里受益良多，结识了许多正能量的朋友，这些对我来说都是非常宝贵的财富。每当我回想起曾经在惠普的那段岁月，心中总是充满了感恩和美好的记忆。

鸟随鸾凤飞腾远，人伴贤良品自高。要感谢的人太多太多，是你们这些良师益友让我一直在荆棘中不断前行，采拾鲜花。

最后还要感谢电子工业出版社王二华编辑，在我写作期间给予了很大的帮助和指导。

专家推荐

近几年，微服务的兴起对从事资讯科技的各级别、各功能、各种技术的人员来说是一重大挑战，也是一个突破的机会。本书深入浅出，让读者很容易就了解到什么是微服务。另外，本书也让已经对微服务有一定认知的读者了解到微服务如何可以应用，在什么环境、什么情况下，会面对什么困难和问题，如何通过不同方法加以解决，让微服务可以顺利地应用，让微服务所带来的好处可以大大地体现出来。对微服务感兴趣的读者和对微服务的应用感到困惑的读者，以及对微服务的场景应用需要更深入了解的读者，可以阅读这本书，不可错过。

电讯盈科解决方案中国区总经理　潘家驰（William Poon）

软件架构模式的衍变往往有其深刻的历史和业务背景，而在互联网+、云计算、大数据、区块链等新的商业模式及相关技术体系大行其道的今天，微服务模式很好地适应了时代的发展，其辅以相应的管理流程、体系，实践了 DevOps 理念，为 IT 产业的蓬勃发展奠定了技术基础。

市面上关于微服务的书浩若烟海、不胜枚举，本书的作者从实际工作的丰富积累出发，清晰地描述了微服务的“微”、“独”、“轻”、“松”等特点的具体实践。针对企业应用微服务模式中遇到的选型、服务拆分、开发技术及相关工具等重点、难点做了阐述，配合实际生产案例，便于读者快速理解。

作者是我在惠普时的老友、多年的同事，一直在一线从事软件研发和项目实施相关工作，具有丰富的软件架构设计经验。本书将给有志于规划、设计微服务模式的架构师、软件设计和开发者们提供一条可行的路。

云星数据（深圳）有限公司　副总裁兼研发中心总监　楼炜

随着技术的不断发展，企业使用的软件功能变得越来越强大，开发维护的难

度也呈指数级上升。如何高效率地开发一个庞大系统，同时做到后期维护成本低，是每一个架构人员考虑的问题。Dubbo 的分布式服务架构，让大家更多地关注了微服务。在大型的互联网电商平台，需要应对极高的并发场景，分布式服务贯穿其中，有成百上千的应用实例在对外提供着微服务。

在传统的企业，由于没有这样的高性能要求，对于微服务的意识并不强。公司前段时间招聘一批技术人员，应聘者很多，但懂微服务的人却很少。只有一些有电商背景和喜欢钻研技术的人，才对 Dubbo+Zookeeper 有比较深入的了解。我很是感慨，公司很重视微服务技术，但外面用的又不多，希望借此书也能将微服务做一些推广。这是一种潮流，需要我们去引领。不过还好，技术的领导者已经嗅到了微服务未来的蓬勃发展，在 Spring MVC 的基础上又推出了 Spring Boot，简化了 REST 服务开发。后期又推出了微服务架构 Spring Cloud，打通了微服务的全流程。这对于想了解微服务架构，或者想步入互联网公司的人员来讲，无疑是一个福音。

有了技术领导者的带路，还必须有大量的推广者，才能使一项新技术发扬光大。本书作者纪晓峰就是微服务技术的发烧友，一个坚定的推广者。为了普及微服务知识，特意撰写了这本书。他是我 HP 的同事，工作认真，性格开朗，项目经验丰富，对 Spring Cloud 的微服务框架有很深的研究，能够用通俗易懂的语言让读者在短时间内熟悉微服务。

相信读者看了此书之后，一定能被作者的行文风格所吸引，在潜移默化中加深对微服务的认识。读此书，就如同读小说，又如同读人生。你会同时收获到微服务以外的很多其他知识。

京东商城架构师　孙海波

目　录

第一部分　微服务解惑篇

第 1 章　微服务架构……2
1.1　不微你就 OUT 了……2
1.2　产生背景……3
1.2.1　时代的召唤……3
1.2.2　容器神助攻……4
1.2.3　微服务的进击……4
1.3　微服务的标签……5
1.3.1　单一职责……6
1.3.2　微……6
1.3.3　面向服务……7
1.3.4　自治……8
1.3.5　易扩展……9
1.3.6　流程化……10
1.4　实力碾压……11
1.4.1　组件化……11
1.4.2　快速……13
1.4.3　可复用……13
1.4.4　机动灵活……13
1.5　不要奶我……14
1.5.1　不足……14
1.5.2　挑战……15
1.5.3　小结……16
1.6　本章小结……17

第 2 章　为何选择微服务……18
2.1　传统架构的病……18
2.1.1　中年危机……18
2.1.2　宝宝心里苦……19
2.2　微服务有药……21
2.2.1　讳疾忌医……21
2.2.2　朋友听说过安利么……22
2.2.3　沙盘演练看效果……22
2.3　微服务的价值……23
2.3.1　资源价值……24
2.3.2　业务价值……24
2.3.3　技术价值……24
2.3.4　用户价值……25
2.3.5　未来价值……25
2.3.6　小结……26
2.4　定个小目标……26
2.4.1　持续交付……26
2.4.2　业务敏捷……26
2.4.3　独立演进……27
2.4.4　高可用……27
2.4.5　高性能……27
2.4.6　站在云端……28
2.5　别人家的公司……28
2.6　来不及解释快上车……29

第 3 章　我拆我拆我拆拆拆……30
3.1　拆还是不拆……30
3.2　如何拆……32
3.2.1　庖丁解牛……32
3.2.2　数据模型和业务模型…33

3.2.3 金字塔结构图 ……33
3.2.4 关键指标 ……33
3.3 粒度 ……35
3.3.1 五笔字根 ……36
3.3.2 被遗忘的字典师傅 ……37
3.3.3 小结 ……37
3.4 边界 ……37
3.5 本章小结 ……38
第 4 章 如何使用微服务 ……39
4.1 如何规划 ……39
4.2 微服务重构 ……42
4.2.1 影响因素 ……42
4.2.2 重构原则 ……43
4.2.3 重构方法 ……44
4.3 本章小结 ……45
第 5 章 微服务的朋友圈 ……46
5.1 好朋友容器 ……46
5.1.1 容器的来历 ……46
5.1.2 容器的好处 ……47
5.1.3 小结 ……49
5.2 天生一对 DevOps ……49
5.2.1 DevOps 是什么 ……49
5.2.2 为什么采用 DevOps ……50
5.2.3 小结 ……51
5.3 SOA 是谁 ……51
5.3.1 接着忽悠 ……52
5.3.2 SOA 是个什么鬼 ……52
5.3.3 小结 ……53

第二部分 技术实现篇

第 6 章 Spring Boot ……56
6.1 介绍 ……56
6.2 特点 ……57
6.3 Spring Boot 和微服务之间的关系 ……57
6.4 如何使用 ……58
6.5 可定制化 ……61
6.6 本章小结 ……64
第 7 章 Spring Cloud ……65
7.1 Spring Cloud 优势 ……66
7.2 Spring Cloud 特点 ……67
7.3 Spring Cloud 之服务管理 ……68
7.4 注册中心 Eureka ……69
7.4.1 什么是注册中心 ……69
7.4.2 为什么要有注册中心 ……69
7.4.3 Eureka 来历 ……70
7.4.4 Eureka 机制 ……71
7.4.5 如何使用 ……72
7.5 网关 Zuul ……77
7.5.1 Zuul ……78
7.5.2 反向代理 ……78
7.5.3 如何使用 ……79
7.6 客户端负载均衡 Ribbon ……84
7.6.1 负载均衡 ……85
7.6.2 如何使用 ……85
7.6.3 补充说明 ……91
7.7 断路器 Hystrix ……92
7.7.1 如何隔离 ……93
7.7.2 电路熔断器模式 ……93
7.7.3 回滚降级 ……94
7.7.4 如何使用 ……95
7.7.5 交通疏导 ……101

7.7.6 小结 ····· 101
7.8 分布式配置中心 Spring Cloud Config ····· 101
7.8.1 为什么要有配置 ····· 102
7.8.2 集中配置的好处 ····· 103
7.8.3 配置方式的演进 ····· 103
7.8.4 如何使用 ····· 104
7.9 服务之间调用 Feign ····· 114
7.9.1 Feign 如何使用 ····· 114
7.9.2 补充说明 ····· 118
7.10 服务追踪 ····· 119
7.10.1 Spring Cloud Sleuth 介绍 ····· 119
7.10.2 Sleuth 如何使用 ····· 120
7.10.3 日志聚合 Zipkin ····· 122
7.10.4 小结 ····· 127
7.11 Spring Cloud 和 Dubbo 的比较 ····· 127
7.11.1 生的伟大 ····· 127
7.11.2 死的憋屈 ····· 127
7.11.3 没比较没伤害 ····· 128
7.11.4 现状 ····· 128
7.12 Spring Cloud 与 Spring Boot 的关系 ····· 129
7.13 本章小结 ····· 130

第 8 章 其他相关技术和工具 ····· 131
8.1 数据库开发持续集成工具 Liquibase ····· 131
8.1.1 特性 ····· 132
8.1.2 使用 ····· 136
8.2 Swagger ····· 144
8.2.1 为什么使用 Swagger ····· 144
8.2.2 如何使用 ····· 145
8.2.3 小结 ····· 149
8.3 权限 spring security ····· 150
8.4 微服务架构的通信方式 ····· 152
8.4.1 同步 ····· 153
8.4.2 异步 ····· 156
8.5 服务编排 ····· 160
8.6 管理工具 JIRA ····· 162

第 9 章 测试相关 ····· 166
9.1 单元测试 ····· 166
9.1.1 Mockito 如何使用 ····· 168
9.1.2 Mock 与 InjectMocks 的区别 ····· 171
9.2 接口测试 ····· 173
9.3 代码质量管理工具 Sonar ····· 174
9.3.1 特点 ····· 174
9.3.2 使用 ····· 175

第三部分 项目实战篇

第 10 章 三个典型系统案例 ····· 188
10.1 企业运营支撑系统 ····· 188
10.1.1 背景 ····· 188
10.1.2 现状 ····· 188
10.1.3 原因调查 ····· 189
10.1.4 解决方案 ····· 190
10.2 车队管理系统 ····· 193
10.2.1 背景 ····· 193
10.2.2 技术架构 ····· 193
10.2.3 服务如何拆分 ····· 194

10.3 财务管理系统……195
10.3.1 背景……195
10.3.2 现状……195
10.3.3 原因调查……196
10.3.4 解决方案……196
10.4 本章小结……197
第 11 章 开发管理……198
11.1 管理原则……198
11.2 日常管理……198
11.3 代码质量管理……199
11.4 工作方式……199
11.5 BA 的职责……199
11.6 SA 的职责……200
11.7 DEV 的工作原则……201
11.8 本章小结……201
后记……202

第一部分

微服务解惑篇

一说起微服务，很多人都是满脑的问号，具体如下：

- ✧ 微服务是什么？
- ✧ 我为什么要选择微服务？
- ✧ 微服务能带给我什么？
- ✧ 我该如何使用微服务？

第一部分通过一套完整的知识体系，让读者对微服务有一个全方位的了解。读者有了这把钥匙就可以解开心中疑虑。



微服务架构

1.1 不微你就 OUT 了

现在众多国外大型互联网公司和传统型公司都使用了微服务架构，这里面比较典型的有亚马逊、沃尔玛、eBay、Spotify、Netflix 等。国内的一些公司，包括一些互联网公司和世界五百强企业也已经投入轰轰烈烈的微服务工作之中。

如果你还不知道微服务是什么，你就 OUT 了。

微服务架构就像葫芦娃兄弟，每一个葫芦娃都有自己的能力果实，随便拎出一个来都可以独当一面，大力果实、火火果实、隐身果实等。每个葫芦娃都是能力单一且唯一的，组合在一起就是无所不能的葫芦小金刚，然后就如意如意随我心意了。

最能体现微服务架构思想的就是活字印刷，二者有异曲同工之妙。

传统的雕版印刷，经常出现一个字刻错了，整个雕版都作废的情况，耦合度非常高，很像传统应用里一个错误就拖垮整个应用的情况。

后来毕昇为了解决这种问题放了一个大招，发明了活字印刷。“毕姥爷”的活字印刷闻名于世，每个胶泥制字都是一个独立的组件，再通过组合的方式完成一套完整的印刷版面，实现了完全彻底的解耦合，再也不会因为一个字刻坏了，而废弃整个雕版的情况。

活字印刷术的发明，不仅非常环保，减少了浪费，而且降低了时间成本、物

料成本、印刷成本，极大地提高了生产力，缩短了交付周期，更是改变了历史，为人类文明的发展做出了巨大贡献，影响深远。

毕昇是世界上最早的架构师，向“毕姥爷”致敬。

微服务就像活字印刷一样，每个服务都是一个组件，通过编排组合的方式来使用，从而真正做到了独立、解耦、组件化、易维护、可复用、可替换、高可用、最终达到提高交付质量、缩短交付周期的效果。

随着架构新时代的到来，是选择上微服务的船驶向新世界还是偏安一隅？生活不止眼前的苟且，还有诗与远方。

1.2 产生背景

微服务架构是互联网行业高速发展的必然结果，传统应用架构已经在持续增长的业务需求、日益激烈的竞争、瞬息万变的市场情况下独木难支，纵然使出“洪荒之力”也枉然。

面对更高要求、更快交付、更多挑战的市场，为了达到敏捷、精益的目标，业界迫切需要一种新的架构能够力挽狂澜，接棒传统应用，继续前行。

微服务踏着七彩祥云就出现了。

随着传统行业也加入微服务大军，微服务架构不再只是互联网行业的专属，大部分传统行业也可以采用这种架构。随着应用范围的逐步扩大，微服务将以星星之火可以燎原之势大放异彩。



1.2.1 时代的召唤

It was the best of times， it was the worst of times.

狄更斯笔下的时代已经被历史车轮远远地甩在身后，现在的年代是最快的年代。

这不仅是大鱼吃小鱼的年代，更是快鱼吃慢鱼的年代。

行业竞争激烈，需求增长快，变化快。面对竞争，快才是硬道理。

业务领域需求变更频繁，如销售、市场、营销都对系统提出了越来越高的要求，需要快速上线，缩短上线周期。

当近期北京打掉三个涉黄场所之后，骗子闻风而动，炮制了下面的诈骗短信，如图 1-1 所示。类似的热点事件经常被蹭。

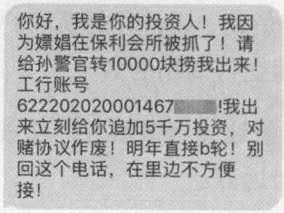


图 1-1　诈骗短信

现在连骗子都知道紧跟时事、蹭热点，更何况企业。

如果不能在瞬息万变的市场情况下做出快速反应，那只能被竞争对手远远地甩在身后。

1.2.2　容器神助攻

在容器技术没有出现以前，微服务所面临的“最后一公里”问题很棘手，因为服务多了以后如何自动化部署、如何运维是头等大事，没有合适的解决方案之前，很多人都对微服务心猿意马。

但是随着容器技术的出现，一切都不同了。

容器技术解决了微服务“最后一公里”落地的问题，从而完成了微服务生态的最后一块拼图。

1.2.3　微服务的进击

微服务概念的提出已经有几年时间，微服务相对于传统应用可以说是脱胎换骨。

人们常说好的架构不是设计出来的，是进化出来的。

非著名相声演员曾经说过，很多相声演员相声说得好，是因为这些演员就是相声世家，父亲就是说相声的，孩子从小受熏陶，所以说得好，是站在父亲的肩膀之上。

软件架构也是这样，任何成功的架构必定是在原有的架构基础之上进化出来的，经过了多少痛苦的挣扎和不懈的努力才能最终破茧成蝶，成为合格的架构、成功的架构，微服务就是在传统架构的基础之上进化而来的。

这一架构体系相对于原来传统的架构模式来说是具有划时代意义的。

从此软件架构翻开了新篇章，驶入新世界。

这就好比毕昇发明的活字印刷术相对于传统的雕版印刷具有里程碑意义一样，无论后来用铜字或者是铅字代替胶泥字，始终都是活字印刷这种模式。

微服务指的是在传统软件应用架构基础之上，按照业务能力将系统拆分成多个服务，每个服务都是一个独立的应用，对外提供一系列的公共服务 API，服务之间以轻量的方式互相调用，如图 1-2 所示。

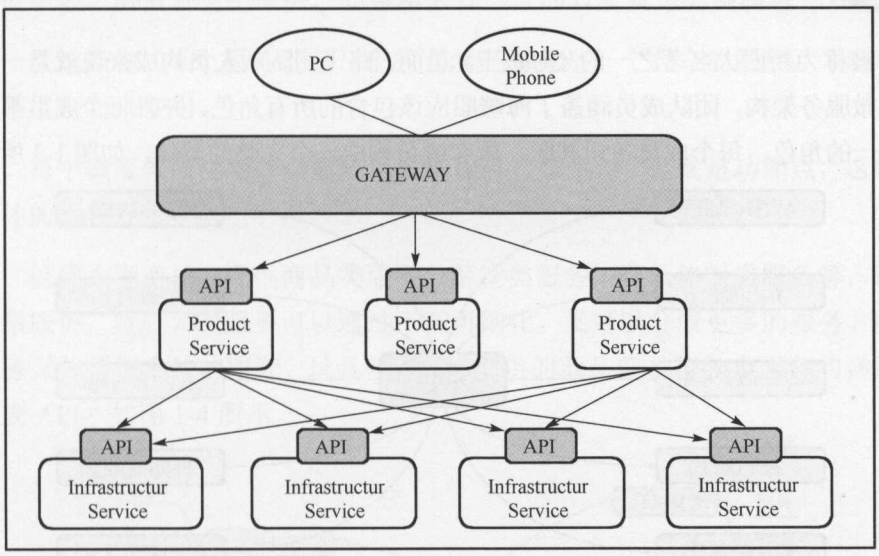


图 1-2　微服务架构图

1.3　微服务的标签

随着互联网进入标签时代，标签已经从线上扩散到线下，生活中的事物大多可以用网络上的标签来概括、进行信息的传递。

微服务崛起于互联网，虽然不是含着金汤匙出生，但也是生来一身标签，很是拉风。

我们来看看它的标签：

- 单一职责
- 微
- 面向服务
- 自治
- 易扩展
- 流程化



1.3.1 单一职责

被称为新四大名著之一的《海贼王》里面，路飞团队从人员构成来说就是一个典型的微服务架构，团队成员涵盖了海贼船应该包含的所有角色，并且每个成员都对应着唯一的角色，每个成员各司其职，所有成员构成一个完整的整体，如图 1-3 所示。

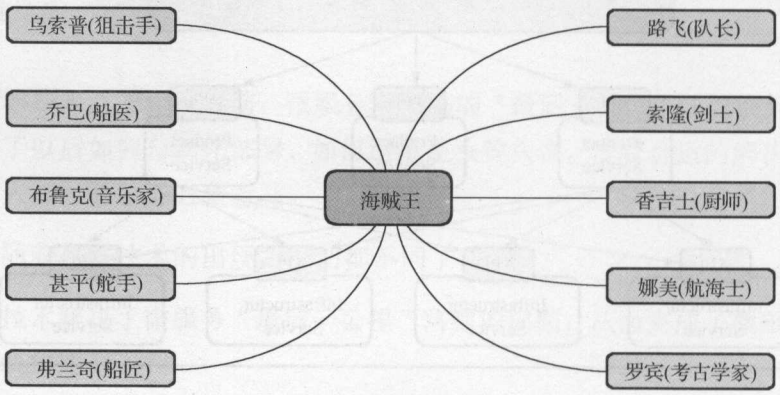


图 1-3　路飞团队

微服务架构当中，整个系统拆分成多个服务之后，这些服务要覆盖所有的业务，并且每个服务都对应着唯一的业务能力，做到单一职责。



1.3.2 微

微是指每个服务都是一只麻雀，虽然很小但是五脏俱全，这个“小”是相对于传统应用的“大”而言的。如果从团队人员构成的角度来说，那么传统应用可

能是几十到上百人的团队，而每个微服务的团队人员很少，如亚马逊的两个比萨理论，就是让人一看，无论从代码量级还是功能数量或者业务复杂度来说，都给人一种一个小项目的即视感。

这就像传统的大型 ERP 系统，如果采用微服务架构进行拆分，其中一个用户服务就可以成为一个独立的应用，因为整个应用都是围绕用户来构建的，业务单一，体量很小，所以用户服务就是个很“微”的应用，当一个系统拆分成若干个服务之后，再将每一个服务进一步细化到 API 级别。



1.3.3 面向服务

微服务的一个重要特征是面向服务，微服务是服务提供者，每个服务对外部提供服务，消费者使用服务，而微服务要达到的效果就是提供满意一百的服务。

在这当中，每个服务既是一个独立的产品也是一个独立的组件，可以和其他服务组合起来使用。

每个微服务应用对外暴露一系列的服务接口 API，也就是功能点，这些都是具体的服务。

就像大型商场，提供商品类服务、餐饮类服务、娱乐休闲类服务等，这些都是微服务，然后大的服务可以通过下沉再细化，又可以分成更多的服务，如商品服务又分了很多种如服装、玩具等，最终提供的商品就是每一个服务的详细功能列表 API，如图 1-4 所示。

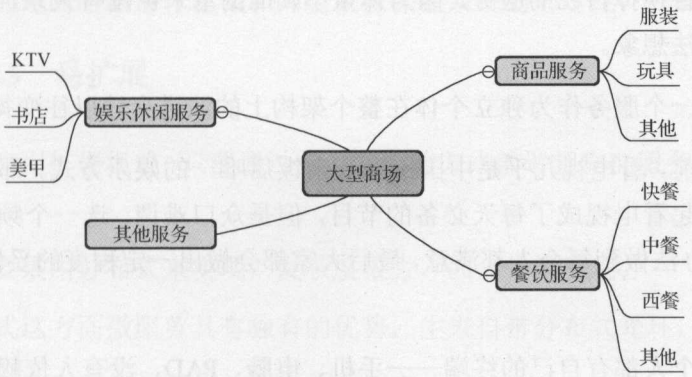


图 1-4　商场服务图

面向服务解除了与平台或语言的绑定，做到了平台、语言无关性，不限定于某一种技术实现，是真正开放的。

而从消费方的角度来看，在使用服务时不需要有任何代码级别的渗透，不需要任何配置，只需要通过 HTTP 直接调用就可以。



1.3.4 自治

从人民公社到包产到户，农民有了自己的一块田，有了选择权，想种植什么作物就种植什么作物，极大地调动了农民的积极性。

以前传统应用是大家在一个没有泳道的泳池里游泳，互相干扰，采用微服务架构之后相当于每个业务团队都有独立的泳道，不会互相影响。

自治包括了以下几方面。

➢ 团队独立

根据业务的拆分，由独立的团队负责一个或若干个服务，团队人数应该限制在 10 个人以内，加班餐正好凑一桌。

➢ 技术选型灵活

独立之后，无论做什么都很自由，不用担心影响其他系统。

每个微服务都有权选择自己的技术栈，只需要考虑是否适合自己的业务特征，以及是否可以解决实际问题，无须关心其他系统，以自己的实现方式来完成，不会出现牵一发而动全身的现象。

还可以替换掉自己的服务，因为体量小，推倒重来也没有关系，这在传统应用上简直无法想象。

因此每一个服务作为独立个体在整个架构上的演进会较以往变得更快。

很多年前，看电视几乎是中国家庭茶余饭后唯一的娱乐方式，那个时候一大家子坐在一起看电视成了每天必备的节目，但是众口难调，选一个频道肯定有人不爱看，没办法做到每个人都满意，最后大家都会做出一定程度的妥协，选择《新闻联播》。

现在每个人都有自己的终端——手机、电脑、PAD，没有人依赖电视了，可以选择自己的娱乐方式，互不干扰。

社会在发展，时代在进步，人选择自己的娱乐，微服务选择自己的实现方式。

➢ 前后端分离

面向服务的架构让前后端分离做得更加彻底，后台提供的服务都是 Rest 服务，与平台和语言都无关，前台框架只要通过 HTTP 请求服务地址就可以获取服务，将返回的 Json 字符串在 H5 上进行展示。因为没有耦合，前端框架可以自由更换。

➢ 数据库分离

对数据库的使用也做到了隔离，每个微服务可以使用自己的数据源，有的用 MYSQL，有的用 ORACLE，非常灵活，更换数据库的时候只需要考虑自己的服务，不用考虑其他应用。

➢ 部署独立

一个微服务重新部署的时候不影响其他服务，局部修改很容易部署，十分利于持续集成和持续交付。

每个微服务的部署都是独立的，不像传统应用，一个地方修改了，需要整个应用重新打包部署，然后另一个地方修改了又要重新部署。导致整个应用被不断地修改、部署。如此频繁的部署非常容易出问题，会影响整个系统的使用，破坏系统的高可用。

➢ 容错

故障隔离，一个服务出现问题不会影响整个应用，传统应用经常因为一个地方出问题就把整个系统拖垮。

微服务通过断路器进行容错和降级，以保证系统的稳定性。



1.3.5 易扩展

传统型应用扩展很难，只能通过全量复制的方式部署到集群服务器的多个节点上来实现。

如果想要采用分布式来实现，复杂度很大，因为先天不足。

在分布式这方面微服务具有独有的优势，生来自带分布式光环。

微服务可以实现局部强化，这点非常厉害。

针对需要扩展的特定服务进行扩展，不需要整体进行扩展，传统应用只能通过整体做集群式的方式来增强，对很多不需要扩展的业务模块来说是一种浪费。

这就像有人想丰胸，如果是传统应用来实现，那么实际效果是脸也大了、肚子也大了，胸是丰了人也“疯”了。

而微服务无论是扩容还是缩容都是有针对性的，定点强化，不会造成要胖一起胖、要瘦一起瘦的结果。

像现在很火的共享单车，相比市政公共交通提供的单车借用服务，共享单车不需要对还车桩的依赖，随借随还，可以无限扩展。而市政公共交通提供的单车由于受限于还车桩的数量，无法提供更多的单车给民众使用，因为要提供更多的车，首先要提供更多的还车桩，还有就是要通过人工方式将资源进行分配和搬运，就要有更多的工作人员进行维护。



1.3.6 流程化

流程化是一套关于用户行为准则的规范，通过采用流程化可以减少犯错的机会，并且不断改进优化。

微服务可以通过流程化将业务难度分解到各个微服务当中，来降低每个服务的业务难度。

好处是方便开发、理解和维护。

开发人员面对的是自己的一块业务，相比于传统应用，业务复杂度极大简化，没有和其他业务的耦合，开发人员可以专注于自己的工作，不用花很多精力在不相干的业务上。

程序员真正做到见山是山，见山还是山，不会迷失在代码森林。

流程化可以带来很多好处，但是如果过于迷信流程的力量，就可能把走流程当成了工作本身，会导致办事效率低下、机构臃肿、官僚作风盛行。

这也就要求在使用流程化时要考虑影响，不能为了使用流程而使用流程，流程应该是为了提高生产率而使用。

两个比较典型的流程化例子如下。

- 造枪

在美国南北战争时期，对于枪支的需求旺盛，但是受限于手工制作。按照传统思维，如果只是增加工人，加班加点仍然无法满足市场需要。

后来厂家将枪支的生产拆分成多个环节，每个工人负责制造一个零部件，再由其他工人进行组装，这样一支新枪就制造完成了。

随后通过这种流水线作业的批量生产，把造枪的过程简化为若干工序，每一组成员只负责一个工序，每一个零部件都按统一的标准生产。

这样无论是生产效率还是质量都大幅提升，生产成本也大幅下降。

- 富士康

一台 iPhone 很复杂，如果问富士康的工人有多少零件，工人肯定不知道，但是由于分解到不同的车间，工人在流水线上只要负责组装零件、拧螺丝，做好自己的一块工作就可以了。

如果系统也能做到流程化和标准化，每个人负责自己的一块业务，程序员只在这个范围内做开发和维护，只要遵循流程和标准就好，这样既可降低难度，又可避免影响扩大。

程序员如果像 KFC 的员工，做的都是流程化和标准化的工作，这样最容易保证代码质量，不出问题，质量如一，品质有保证。

利用微服务的无状态性，实现流程化和标准化，虽然可以降低每一环节的复杂度，但是不可能降低事物整体的复杂度。

1.4 实力碾压

从微服务的标签就可以知道它绝非池中之物，不用看气质，而是靠实力，从各方面碾压对手，下面来看看它的技能点。

- 组件化
- 快速
- 可复用
- 机动灵活



1.4.1 组件化

微服务带来了组件化的好处，就是组件的独立、可复用性、可替换性等，每

个组件就像活字印刷里的每个活字一样。还有我们比较熟悉的乐高玩具也是最具有组件代表性的。

不同于传统的 Jar 包形式的组件化，服务化的组件化在耦合度上更低，如果 Jar 包改变了需要重启整个环境，而服务化的情况下，每个服务运行在不同的进程中，独立的服务的变化只需要重启这一个服务即可，实现了影响最小化。

微服务是组件化最彻底的架构，每一个服务是高内聚的组件，业务功能集中，对外则是低耦合。

想要把家里收拾得整整齐齐、有条理，这不仅需要有很好的整理分类能力，更需要一些家具和收纳箱的支持。

如果衣柜本身没有挡板，将它按使用功能分隔成独立的一系列空间，所有物品都往里面摆放，就成了传统应用的样子，任你如何归置都无法做到易放易取。

这时就需要配合使用一些收纳箱，将同类物品归类之后统一存放，这样就做到了更细化的隔离。

每一个收纳箱都是一个独立组件，组件之间互不干扰，想拿内衣时不会把袜子取出来。

微服务通过组件化专注于单一业务领域，不会牵扯其他业务域。通过隔离，哪有问题改哪里，和其他不相关的业务无关。

事实上不只是软件工程存在着耦合、依赖的关系，机械工程同样如此。

在大型机械制造领域，如北欧的重型机械，一般不容易坏，坏了也好修，哪里坏了拆哪里，同时还有很好的监控，记录所有的操作，知道是怎么坏的，定位问题也更容易。

好的架构就要做到易维护。

微服务通过组件化的方式做到有针对性的修复，最大程度地做到易维护。

而反观国内的品牌，修理的时候难度特别大，很多零件都耦合在一起，要拆一个地方经常要把其他不相干的模块一起拆下来，不然没法修，和传统应用很像。

这就是设计理念上的差异，在设计的时候应该先考虑坏了以后如何修理的问题。

北欧的制造业如此发达，小到乐高大到宜家，全都是组件方式，组件化的优点就是耦合度低，可拆、可换、可复用。



1.4.2 快速

活字印刷较传统印刷交付快的一个主要原因是传统印刷一次要刻整板，活字印刷只刻个别字，这就像传统应用的开发工作量特别大，导致开发效率低，而微服务要开发的工作量小，所以开发速度快。

相对于传统应用的重装重甲配置，微服务就像是轻骑兵，以反应快、速度快著称。

在交付效率方面，微服务要比传统应用快得多的多，充分发挥了小、快、灵的特性。

从体量上来看，一个服务只有原来系统的 N 分之一，所以无论是开发、测试还是部署都非常快。



1.4.3 可复用

因为组件可复用，活字印刷较传统印刷交付速度快很多，减少了开发时间，以及因为失误而造成的返工时间。

微服务的可复用性也可以帮助它更快地开发和交付。

微服务当中，服务之间没有绑定关系，每个服务都具有很好的可复用性，通过简单地设置变成另一个服务。就像活字印刷里的活字一样，可以重复使用，避免了重复性的工作。

像酒店的门卡，它与房间没有耦合关系，上面没有房间号，提高了安全性，同时任何一个门卡都可以通过简单设置变成任何一个房间的钥匙，没有依赖关系，可以无限复用。

而传统应用是一把钥匙开一把锁，可复用性非常低。



1.4.4 机动灵活

可以把传统应用想象成一支军队，而微服务则是化整为零，变成多兵种陆海空协同作战。

一支军队的好处是可以集中火力，但是机动性不好，在面对分散的战线时容

易顾此失彼，一旦被打开一个突破口，容易被全部歼灭。

一个系统上某一处的问题，都可能导致故障的漫延，最后拖垮整个系统。

多兵种协同作战的优势是机动、灵活，可以多线作战、四处出击，又可以协同作战，当一个军种被歼灭之后，还有其他后备军可以顶上，保证有生力量。

微服务架构中当一个服务“挂掉”了，还有其他的服务可以正常使用，同时可以立刻启动服务的其他实例作为后备部队。

1.5 不要奶我

当有些人过于夸大微服务时，我们一定要表明自己的态度：这口毒奶我们不干。

干了之后，微服务就危险了，所谓希望越大，失望越大。

对待微服务一定要保持客观的态度，诚然微服务能在大多情况下带来很多好处，但如果指望微服务可以解决一切问题又是不现实的。

所谓术业有专攻，选对方向、挑对了路才是王道，适合自己的才是最好的，某些情况下传统应用也很好。

使用微服务时我们也要知道它的不足之处，以及所面临的挑战，只有清楚这些以后，才能更好地使用微服务。



1.5.1 不足

➢ 时效性

微服务不适合所有的业务场景，如时效性高的系统，这方面不是微服务的长处。因为它本身分布式的特点，在拆分成多个服务之后，服务之间的调用会有额外的开销，当部署在不同平台或网络时，服务间的调用延时可能导致系统响应慢的问题。

➢ 一致性

基于分布式事务的复杂性，微服务在保证事务强一致性方面需要做更多的工作，虽然理论上可以做到，但是显然这并不是它最擅长的。

➢ 需要投入

天下没有免费的午餐。要想享受新架构带来的好处，就要有投入。

➢ 可能造成浪费

各个服务之间可能形成一座座孤岛，这样会造成一些资源的浪费，如一些重复开发的功能。在很多服务中可能都会使用到同一个功能，而这一功能没有足够大到成为一个服务的程度，这个时候可能不同的服务团队都会单独开发这一功能，重复的业务逻辑，违背了良好的软件工程中的原则。

这种重复开发的情况就像在云计算和大数据兴起时一些公司内部很多 team 都在各自做各自的研究，资源和成果都没有实现共享，浪费了很多人力和物力。

但是长期来看，微服务更像是一次性投入长期受益的理财项目，虽然会一次性投入很多资金、时间和精力，但是一劳永逸。



1.5.2 挑战

不经一番寒彻骨，怎得梅花扑鼻香。

想要从使用微服务中获益，在使用的过程中遇到各种挑战是无法避免的。这时如果没有充分准备，可能会措手不及，下面让我们来看看都有哪些挑战，提前做好准备来应对。

挑战主要来自三个方面。

第一个是落地。

这里边包含了 PAAS 等基础设施，还有 Docker 等环境的准备工作。这类基础设施对于一些公司来说可能早就具备了，但是对于某些团队来说，具备这些基础设施可能需要更多的支持和时间。

第二个是规划。

如何整体规划是个艰巨的任务，规划不合理就会返工重做，造成重复投入、重复建设等不良后果。规划包括技术选型和梳理业务。

微服务的实现技术有很多，要根据不同业务的特性选择适合它的技术，在不同技术之间进行比较、取舍，保证系统能够顺利实现。

梳理业务这部分工作也是非常重要的，倘若系统承载的业务比较复杂，那么

将所有的业务进行全面、清晰、正确的梳理则是一件相当有挑战性的事情，如果梳理业务的工作没做好，就会给后续工作带来很多麻烦。

第三个是开发、测试和运维。

这部分挑战比较多，主要体现在以下方面。

（1）接口规范

因为微服务面向服务的特点，所有对外的服务都是以接口形式存在的，因此需要设计出兼容性强的接口，以避免因为服务变更导致消费方无法调用的情况发生。服务方如何做到来者不拒、宽进严出，这些都需要定义良好的接口规范来支撑。

（2）契约一致性

微服务架构的多个产品服务会依赖于基础服务。当契约变更时，如何保证调用双方契约一致性是个问题，否则将出现服务不可用的情况。

（3）服务编排

随着拆分的粒度变细，服务的数量也会越来越多，服务间的调用关系会更加复杂，这就需要有专门的服务编排工具进行处理。

（4）容错

在服务调用方面，因为服务数量多，难免会发生被调用服务不可用的情况，这时如何更好地做到服务降级和容错是比较大的挑战。而当其中一方服务不可用时，消费方如何优雅地处理错误也是一个考验。

（5）运营管理

拆分成多个服务之后的管理工作对运维团队是个挑战，在使用微服务的过程中为了更好地管理所有的服务，需要引入很多辅助工具，如日志、监控等。

在服务管理方面，会随着服务数量的增加复杂度也不断增加，在多服务的配置、部署和资源调配方面需要有高度自动化的持续集成、持续部署环境。一个传统应用只需要部署在服务器上就好了，而部署一个微服务应用需要掌握一整套的部署工具。



1.5.3 小结

经常发现一些本来很好的东西，但是一到国内应用起来立刻面目全非，如保险、直销、投资理财工具等，这种现象叫南橘北枳。

微服务本来是个好工具，但是如果使用不当可能也会变成“北枳”。能否使用好这个工具的影响因素很多，如果不能很好地解决这些问题，那么再好的工具也会被浪费，最后怪罪工具的事情也是常有的。

为了避免微服务重蹈覆辙，在选用时要慎重，采用什么技术架构，需要做哪些准备，如果规划设计得不好，可能会导致重构，影响业务正常的使用，这时微服务就要“背锅”了。最后可能会得出一个结论：微服务还不如传统应用。

1.6 本章小结

本章介绍了微服务架构是什么、产生的背景、它的特点和优势，以及它的不足和挑战。

通过这一章的介绍读者可以对微服务有个初步的认识。

下一章介绍为什么要选择微服务。



为何选择微服务

选择微服务总得需要一个理由，好不好不是绝对的，是相对的。首先看它和传统架构的比较，其次是看它能带来哪些收益，最终通过它能达成什么目标，再来参照“成功人士”的选择，然后就可以放心使用了。

2.1 传统架构的病



2.1.1 中年危机

岁月是把杀猪刀。

小鲜肉如何变成猪肉荣？一个普遍现象：多年后的同学聚会，男生普遍都胖了，肥头大耳，挺着大肚子。曾经的小鲜肉就这样变了样。变化如此巨大，让人始料不及。胖了之后，身体不灵活了，身体小毛病也多了，总之就是整个人都不好了。

很多传统应用大抵是这个样子，刚上线时意气风发，各功能模块划分得很好，自带六块腹肌。

经历了多年的发展，项目承载了更为复杂的业务，以及激增的用户数量，随着业务需求的指数级增长、变化，新功能的不断加入，与多个外部系统对接，这时传统应用一下子从一个了无牵挂的单身小伙子变成了上有老下有小的中年人，各种耦合特别多。

在核心人员流失更迭、自身架构不足等各方面因素综合影响下，系统越来越糟糕，代码逻辑难以理解，开发难，易产生 BUG，为了解决 BUG，产生更多 BUG，各种补丁让代码的可读性更差了。

为了解决问题引发更多的问题，这样的恶性循环呈螺旋式放大。最直观的感受，系统变慢了，反应和电影《疯狂动物城》里的闪电一样。



2.1.2 宝宝心里苦

以前一说起传统架构的问题，因为身受其害，都是一肚子苦水，宝宝心里苦啊，宝宝不说。因为说了也没有用，现在不同了，因为有其他选择方案。

先来倒倒苦水，说说传统架构的那些问题。

➢ 笨拙

有人把传统应用称为单体应用或巨石应用，巨石叫法虽然不太贴切，但也突显其笨拙的一面，不知如何下手，难以搬运。

无论开发、测试还是部署都非常慢，导致交付周期很长。

整个团队像极了伏尔加河上的纤夫，奋力在拉船，然而船仍然行驶缓慢，完全是体力活儿，让纤夫没有爱。

➢ 耦合严重

业务与业务之间耦合在一起，代码与代码之间耦合在一起，导致开发、维护困难。

耦合严重还带来一个问题就是 BUG 藏得很深，不容易被发现，定位问题困难，修改也困难，牵一发而动全身，要了解整个系统才能进行开发，否则无法下手。

同时因为所有模块都运行在同一个进程中，任何一个模块中的 BUG，如内存泄露，都将会拖垮整个进程。这种耦合现象，就像雕版印刷因为一个字错了而废弃整个板子一样。

因为所有业务模块都被绑定到一个应用上面，其中一个模块做了修改，上线时就要把整个应用停掉，这样会影响其他业务模块的正常使用，就相当于一个团队只有个别人员需要加班，但是全部成员都要留下来一起陪着加班一样，非常不合理。

当原有系统要集成一些新的功能、引入第三方的框架之后，经常导致代码的严重侵入，如开发 WebService 时引入 Axis 的工具，需要生成额外的大量的客户端代码，对原有系统是一种代码污染。

➢ 协作困难

所有人都针对一个应用进行开发，当人数增多之后，互相影响，对于核心模块的修改会导致各种冲突不断，当多人同时操作时，代码很容易被不小心覆盖掉，进而引发严重的问题。以前很容易完成的功能，现在开发和测试的时间都变长了。

➢ 无法持续交付

所有的功能都依赖于一个大应用之上，应用的体型就像是变异的怪物哥斯拉一样，应用与应用之间的依赖导致部署困难。

每次上线涉及多个应用的部署，有一个配置没有提交或者某个环节出问题都会导致部署出错，影响上线的正常进行，每次上线都是一场大考。因为应用的部署依赖太多，不可控因素太多。

➢ 受技术栈限制

技术选型一旦确定，那么所有成员都只能按照这种技术来进行开发，没有回旋的余地，再招聘的时候也只能按照现在的技术要求进行招人，导致技术方面特别单一，抑制了技术的多样性，当出现特别适合某一模块的技术时也无法进行技术升级换代。

➢ 债务多

生产问题多，运维不断地报出各种问题。开发人员每天来了先要解决生产问题，就像出租车司机每天一睁眼就欠了一笔份子钱一样，就像房奴每天一睁眼就欠了银行一笔房贷一样，因为大量精力都用于解决生产问题，导致开发时间被大幅压缩，这样的开发质量可想而知，并造成一轮又一轮的恶性循环。

在传统应用中很常见，因为需求的变更，技术上有时会向业务妥协，设计不合理的地方也常有，往往是先把功能做上去，计划以后再重构，结果重构的日子遥遥无期，转眼开发人员各奔东西，最后都成了债务，而曾经的承诺大多成了“白条”。

另一方面任何一个代码都可能被很多地方引用，很难修改，有时候代码看似不合逻辑，也改不了，这些债务也导致很多新来的“接码侠”融入慢、学习周期长，难以快速上手开展工作，极大地降低了工作效率，造成了浪费。

2.2 微服务有药

微服务：你有病啊？

传统架构：你有药啊？

微服务：你吃多少？

传统架构：你有多少我吃多少。

面对传统应用的大部分病症，微服务的存在如同一剂良药，让大家看到了希望。



2.2.1 讳疾忌医

很多人都会犯讳疾忌医的错误，在病情轻微时忽视，进而引起更严重的病变，最终无药可救。

“君有疾在腠理，不治将恐深。”这是一个关于时机的问题，趁早治疗还有的救，因为治疗是要考虑很多因素的。微服务虽然可以解决传统应用的大部分问题，但是它终究是一个工具，无法起死回生。

传统应用的痛就像牙疼，虽然明知早治早好，但是就是下不了决心，觉得还可以忍受，最终发现还是忍不了，然后去拔牙，悔不如当初早点治疗。

这是一个单向不可逆的过程，最终结果就是传统应用实在挺不住了，花重金重新来做。

要有合适的人在合适的时间通过使用微服务，对传统应用进行治疗才行。

最佳治疗时机是当还有人对业务和技术架构熟悉的时候。如果已经没有人熟悉系统，那么要多花大量的精力和时间来熟悉系统才能下手。

传统架构或多或少都会有一些问题，如果及时治疗，成本和代价都会比较小。随着时间的推移，越往后越难治，晚期就是“植物人”。

一个系统经数人之手开发了数年，现在要重构，可能重构的成本比重新做一个的成本还要高。

因为随着核心人员的流失，懂业务和对系统熟悉的人越来越少，如果没有一

个很好的知识库体系，那么新来的人对系统的了解只会越来越少，越来越片面。

这时即使投入巨大的人力和时间，最后不一定能治好，只能是死马当活马医。



2.2.2 朋友听说过安利么

好的架构是什么样的？

幸福的家庭都是一样的，不幸的家庭各有各的不幸。

这个说法用在架构上也一样。

当初设计合理的架构不一定满足未来的需要，当初设计不合理的架构则注定上演悲剧。

国内很多系统都是紧急上线，短期内用“人肉”堆出来的，埋下了很多隐患，挖了很多坑，领导不管那些，先上线再说，还得用这个当成功案例做推广呢，底下的人也跟打了鸡血一样拼了老命把系统给上线了。

结果就是苦了后来的“接码侠”，谁接谁知道。

好的架构就是一开始就要考虑后面的扩展和修改。

据说以前配毒药的时候，要先把解药配出来，这是为了防止自己中毒“扑街”。

这就像《富士山下》里唱的，“要拥有必先懂失去怎接受”。

好的架构应该是解耦的，如果修改一个地方，不影响其他地方，不需要从前台到后台的联动。如果一个功能要修改就会涉及从前台到后台的一系列改动，那就说明系统耦合严重。

好的架构一定是轻量灵活的。

船小好掉头，可以应对千变万化的业务需求。不会像传统应用那样本身架构对新业务的支持不好，只能让业务和功能互相妥协，勉强把新业务功能加上去。这种妥协是对双方的伤害，既破坏了原有架构，又埋下了隐患。



2.2.3 沙盘演练看效果

传统应用的病不是简单打一针就能看到疗效的，所以论证的过程就显得十分重要。论证可以通过沙盘演练的方式来实现，看看使用微服务之后到底灵不灵。

➢ 公输班和墨子

战国时，宋国面对强大的楚国的威胁，无计可施，随时可能崩溃，这时候墨子来了，和公输班进行了一场沙盘演练，面对各种攻城方法，墨子一一化解。

先不说实战如何，起码理论上要行得通。

那么是否引入微服务也要进行这么一场沙盘推演，现在系统的问题是否可以通过微服务就能够解决。

比如用具体的需求来说明，之前的一个需求涉及修改代码的地方是多少处，工作量是多少，测试的工作量又是多少，交付周期是多久。现在采用微服务之后，有哪些变化，开发工作量和测试工作量减少多少，交付周期、人员数量等都是可以量化的指标。

比如，新增业务、需求变更，微服务如何应对？

一方攻城，一方守城，微服务就是守城的一方，面对各种需求，如何化解，同时与当前系统的区别是什么，有哪些方面的提升？

传统应用“以不变应万变”，结果就是什么都得吃不了兜着走。因为本身没有变化的技能点，属于先天缺陷。

微服务“以万变应万变”，就如同《西游记》中孙悟空与二郎神之间的斗法，优势就是可以进行各种变化、扩展，全是技能点。可以通过自身的变化应对系统可能的各种变化，并保持系统的一个稳定健康的状态，不会像传统应用一样，为了应对业务的变化不得不破坏自身架构的一致性。

就像一个哲人曾经说过：如何你不能阻止环境的变化，那么就改变自己去适应它吧。

微服务架构可以轻松应对需求的变化，从而达到快速交付的目的。

通过采用沙盘演练的方式可以很直观地论证微服务架构是否可以应对复杂变化，解决实际问题，是否可以根除传统应用固有的顽疾。

2.3 微服务的价值

天下熙熙，皆为利来；天下攘攘，皆为利往。

大人的世界只分利弊，小孩子的世界才分对错。如果仅以对错来区分，那么显然微服务是一个正确的选择。但是涉及利弊的时候要复杂得多，考虑的方面也会更多，如通过它所带来的收益是否大于对它的投资，也就是投资回报比如何。

我们从以下几个方面来讨论一下微服务的价值。



2.3.1 资源价值

有些系统面对偶发性的流量洪峰，原有服务器资源无法满足，加了服务器之后这个问题虽然解决了，但是平时就会导致资源过剩，造成资源浪费。

这方面不用举京东 618、天猫双 11 的例子，就说有的地产中介公司，每月月末同一时间所有经纪人都要通过系统录单子，导致系统压力瞬间放大，资源紧张。

通过微服务，可以更充分合理地利用资源，在资源不足时进行扩容，在资源过量时进行缩容，这样就不需要一直占用资源。如果部署在一些云平台之上，通过按需支出更是能省下一大笔费用。



2.3.2 业务价值

有人不想换微服务架构，没有动力的主要原因是因为不知道通过微服务能带来什么可量化的收益。

其实可量化的收益很多，如工作量、人员数量、交付质量、交付周期等。

通过对沙盘演练论证的前后工作量做对比，就可以推导出整个系统采用微服务之后带来的收益。



2.3.3 技术价值

有人说从一种技术架构换成另一种技术架构，如果只是重新实现了一遍，既没有改变现状，也没有带来收益，那么是没有价值的。

这句话初听时觉得有道理，后来再细想想，觉得并非完全如此。如果只是在某个特定时期的一个具体项目来看，那么单纯的技术价值可能没有意义，但是如果考虑未来情况的话，那么技术价值不仅有，而且很大。技术的价值在于积累和可重用，这是对未来的一个投资，现在可能没有带来实际价值，但是如果将来再推广到其他项目上，那么它的价值就体现出来了。

技术是为业务来服务的，如果通过技术可以促进业务发展，同时通过推广可以扩大技术的影响范围，那么技术的价值就有所体现。



2.3.4 用户价值

有人会说，你说的微服务的好处我都知道，但是我现在的架构用了很多年，客户的使用习惯已经养成，改变并非易事。

什么东西都怕比较，俗话说人比人得死，货比货得扔。用户有了比较就会用脚投票，唯有站在用户的角度，做到用户至上，这样才能得用户者得天下。

最初沃尔玛的老板很传统，不喜欢改变，当有员工向他反映引入电子支付时，他表示不需要。后来有一次他去店里视察，发现因为顾客比较多，收银台排着长龙，顾客都需要等很久才能结完账，他意识到传统支付方式的弊端，回去立刻把电子支付引进沃尔玛。

在引入了电子支付之后，顾客结账的时间大大缩短了，客户体验大幅提高。效率提升了，为企业带来了更多的效益。

从最初的纯人工方式，到后来的刷卡支付、扫码支付、近场支付，方式越来越便捷，效率越来越高，客户感知越来越好。现在亚马逊又在实践自动支付，让顾客根本感觉不到结账的过程。

而微服务最明显的收益是用户体验的大幅提高，过去最快一个月上线一次，需求从排期到上线要 1～3 个月的时间，现在可能一天就上线了，这就是绿皮车到高铁的飞越，大大提高了生产力，在瞬息万变的市场上占尽先机，这是多么大的收益。

用过微服务后就知道其和传统应用相比优势是多么明显。



2.3.5 未来价值

重构之后对于业务未来发展的影响，这部分价值最难以估算。

如果当前架构影响了业务的发展，是一个瓶颈，那么通过微服务的方式就可以解决这个问题，带来业务的发展和提升。就像一些足球俱乐部，因为球场容量太小，无法满足市场需求，同时严重影响了收入，这时通过对球场的改造、扩建就可以达到增加门票收入的目的。

通过技术架构的升级，带来对未来业务的提升，最终让业务能够迅猛发展，创造巨大财富。



2.3.6 小结

都说商人无利不起早，有利益的事情总会让人更有动力，不仅仅是对于商人，对于一个公司来说更是如此。

是否选用微服务肯定也要考虑它的利弊，要擦亮双眼，要多比较、多论证。

投资分短期、中期和长期，对于微服务的使用，可以论证的都是短期的收益，而未来的收益多半无法预测和量化，如果眼光放得长远，未来的收益可能更大。

2.4 定个小目标

对于微服务，我们也要有一个小目标，当然不是赚一个亿。

我们的目标很简单，就是构建持续交付、业务敏捷、独立演进、高可用、高性能的系统。



2.4.1 持续交付

持续交付就是将过去多则几个月短则一个月的上线周期模式彻底改变，可以随时上线、随时交付，大幅缩短上线周期，让传统的周期性交付成为历史。

通过自动化工具构建持续集成、持续交付的环境，实现自动化、流程化和标准化。将原有的从需求到上线的一整套业务流程像流动的水一样不断循环往复，弱水三千，随时都可以取一瓢。



2.4.2 业务敏捷

通过快速响应、敏捷开发、持续交付，真正做到业务敏捷。

传统的排期模式有专门的需求审批部门负责需求的收集和审批，然后定好排期，这是一个很漫长的过程。尽管会开辟类似 VIP 通道的快速响应需求，但实质

上仍然要遵循上线周期，没办法做到真正的敏捷，而微服务架构当中更强调将整个上线周期的需求分散到日常，形成每天上线的一个常态模式。

从此需求不再像等待出行的游客，需要按照轮船起航时间统一安排，而是轮船变快艇，来一个走一个。



2.4.3 独立演进

通过隔离，每个服务在保持原有产品能力的基础上在各自业务泳道上进行不断进化。

大多时候不同业务的侧重点不同，所适用的技术也不尽相同，这是由于业务属性决定的，如侧重统计类报表的业务和频繁写数据的业务肯定不同。这时只有通过独立演进才能发挥各自技术上的优点，更好地为业务服务。

小团队的发展速度要远比大公司快，因为从小变大容易，从大变更大就很难，另外大公司因为部门多、流程多，互相掣肘时有发生，都不利于快速发展。

所以微服务里每个团队都像一个创业型公司，本身具备发展的潜力和驱动力，可以在独立演进之路上走得更加坚决彻底，最终走得更快更远。

就像路飞团队成员在香波地被暴君分别送到不同的地方进行强化训练，针对每个船员的弱项进行补强。这样有针对性的训练，让每一个成员都得到了最大程度的进步。

由于每个船员的角色不一样，需要增强的能力是不同的，如剑士要增强的是剑术，机械师要增强的是技术，二者没有任何相关性，如果在一起，如何让两者共同进步？如果大家都在一起，进步肯定是有的，但不会有好的效果。



2.4.4 高可用

通过架构的优化、持续交付、关键业务的隔离、熔断机制的保障，达到业务的高可用、系统的高可用，减少宕机次数和时间，降低宕机的感知度。



2.4.5 高性能

实现大规模存量数据和快速增长条件下的高性能。

高并发、高性能通过自由伸缩来实现。

自由伸缩就是当业务量小的时候，减少服务实例的数量；当业务量大的时候，同时启动多个服务实例，应对流量洪峰，就像开闸泄洪。

如同超市收银台，排队人少的时候开放的收银台也少；人多的时候，收银台全部开放，就可以极大地提高业务吞吐能力。



2.4.6 站在云端

容器所提供的轻量级、面向应用的虚拟化运行环境为微服务提供了理想的载体。

同样，基于容器技术的云服务将极大地简化容器化微服务创建、集成、部署、运维的整个流程，从而推动微服务在云端的大规模实践。

2.5 别人家的公司

亚马逊的选择。

一家伟大的科技公司其成功之处在于做事的方式和态度。

亚马逊这些年的迅速发展得益于他们做事的态度，他们每年会将现有系统做重构，这是需要投入大量时间和精力才能完成的。

他们没有蒙眼狂奔，而是定期停下来做维护和保养，确保航向没有偏离，有时候只有方向选对了，才会在正确的路上越走越远；反之，走得越远，错得越多。

网上有篇文章“亚马逊中国大博弈”，里面介绍了在亚马逊云计算基地落户甘肃中卫的过程中各方博弈、种种曲折。最让人印象深刻的是强势和谨慎。

他们为做一件事光是准备工作就要花几年时间，这如果放在国内的一些公司可能已经做完了。

与亚马逊的谨慎相比，国内的某互联网公司简直到了丧心病狂的地步，真的是蒙眼狂奔，就像那种一生从不停止飞翔的鸟，停下来的时候就是死亡的时候。

亚马逊中国这些年从来不搞烧钱的促销。难道是它没钱么？答案当然不是。它就是保持着自己的风格和态度，不盲从、不跟随。

所以当谨慎的亚马逊都采用微服务的时候，其他人还有什么可犹豫的？

2.6 来不及解释快上车

本章介绍了选择微服务的理由，传统应用的缺陷正突显了微服务的长处，可通过微服务来弥补。

接着为大家展示了微服务创造的各方面价值和目标，最后跟上业界大佬、“成功人士”的脚步，他们的选择成功概率还是大于常人的。

对于系统架构来说，使用微服务可以说是一次升级终身受用。

使用微服务可以赢得未来，但是要趁早。因为当前的系统会越来越大，要还的技术债务也会越来越多。

越早做，工作量越少，可以越早得到技术积累、人才积累，在未来的市场中占据主动。



我拆我拆我拆拆拆

天下事合久必分，分久必合。

“拆”的历史我们比较久远，“拆”的文化我们十分熟稔。

微服务最核心的部分就是业务能力的拆分，这也是为什么要把它从如何使用微服务当中独立出来、单独成一章的原因。

拆分并不是简单的物理拆分，而是一个从具体到抽象再到具体的过程。过程中通常会提取出独立的服务，就像有人身兼数职，现在要将职责拆分出来做到专人专职。

在整个拆分过程中，很多人一开始就针对某个业务来讨论拆不拆的问题。事实上这种做法并不科学，也不会取得理想的结果，就像本章下面要举的这个例子一样。

正确的做法应该是先了解拆分的原则、粒度和边界的问题，然后自然而然地知道哪种是应该拆的、拆成什么样，等等。

3.1 拆还是不拆

To split，or not to split -that is a question.

使用微服务经常会面对这种艰难的决定，拆还是不拆。

拆有拆的好处，不拆也有不拆的好处。

拆有拆的顾虑，不拆有不拆的理由。

这是做微服务时大家都要面对的问题，因为立场不同，经常出现站队的情况，最后只能通过 PK 来决定拆还是不拆。

像下面的一场辩论就很常见。

背景：耦合在一起的两个业务主体，是比较核心的业务，逻辑比较复杂，两者有很大一部分的业务逻辑是一样的。这时候拆还是不拆？

辩论赛辩题：拆还是不拆？比赛开始。

正方：拆。

反方：不拆。

反方：两个业务就放在一个服务当中，也没什么，拆分粒度不应该太小。

正方：微服务的原则就是要把耦合在一起的业务进行拆分，这两个业务是属于比较大的业务单元，从粒度上来说是比较大的，并不会因为拆分导致粒度过细。

如果两个业务耦合在一起，那么开发的时候就要特别小心，改动其中一个很容易影响另一个。改完一个之后要测试，请问另一个要不要测试？

只是其中一个要上线，另一个也要被迫跟着停系统，导致无法使用。

反方：拆分成两个服务之后，原来的大部分代码是共用的，这样如果修改一个逻辑代码，需要同时改两处地方，不符合软件开发的原则。

正方：共用的代码可以提取出来，作为公共方法，这样就不会修改两个地方。

反方：要考虑成本收益的问题，这个拆分之后能带来多少收益？

正方：收益是多方面的，工作量大幅减少，最主要的是业务从提需求到需求上线以前要两到三个月，现在最快只要一天。

反方：暂时先不拆，以后再拆。

正方：系统只要不是处于停止开发的状态，那么系统代码每天都在增长，以

后再拆，难度越来越大。

反方发言：……

万万没想到，最后反方赢了，结论是“不拆”。

因为 PM 说这个业务不在本期计划当中，以后还是会拆的。

正方：以后……

那这场辩论还有意义吗？

这个例子反映了很多人在拆分问题上的分歧，因为很多人都不知道拆分的原则，所以一上来就做这种辩论显得毫无意义。接下来我们用正确的姿势解锁拆分的奥秘。

3.2 如 何 拆

不问拆分的原则，直接问拆不拆的都是耍流氓。

下面讲一下拆分的必要条件、拆分方法及拆分的原则。



3.2.1 庖丁解牛

始臣之解牛之时，所见无非牛者。三年之后，未尝见全牛也。方今之时，臣以神遇而不以目视，官知止而神欲行。依乎天理，批大郤，导大窾，因其固然。

只有对牛的构造特别清楚，才能做到下刀游刃有余。庖丁的成功来自于多年的实践、刷出的经验值。

所以我们要向庖丁学的就是尊重客观规律，熟能生巧。像庖丁解牛一样去拆分就是要我们对系统熟悉，这是做拆分工作的必备条件。无论对整体的架构还是对于具体的业务都要了解，这样才能知道业务功能的划分、模块之间的边界，才能做到条理清晰的拆分。如果只是片面的了解，那就是盲人摸象，拆的是错的。



3.2.2 数据模型和业务模型

那如何才能做到对系统胸有成竹呢？我们可以通过数据模型和业务模型对系统进行梳理，利用这两个利器最终得到一个比较全面的业务范围。

数据模型是骨架，业务是皮肉，所有的业务功能都是依附于数据模型之上的。

传统应用数据实体聚合在一起的大多属于同一业务功能（如常见的客户服务功能、订单服务功能等），在一些工具上（如 PD 图）看数据实体的关系就像分散的一个个部落，可以很直观地看出都有哪些业务功能模块，这些功能模块就可以作为拆分的对象。

通过业务模型可以识别出不同的业务主体的流程环节，将业务主体进行分类，不同类别的业务主体之间通过业务泳道进行分隔，避免互相影响。这样就会得到一个业务泳道图，与后面的金字塔结构图结合起来使用，以此为依据进行业务主体的拆分工作。



3.2.3 金字塔结构图

金字塔结构图用来勾勒出整个系统的所有业务功能点，将整个系统的业务功能按照金字塔的结构划分好，从上到下，分成多个层级，从大到小，每个层级可以下沉、细分，最底层就是最小的粒度，最终得到一个覆盖全部功能的结构图。

这时会发现有很多共同的业务功能，如校验服务、用户服务、配置服务等，这就需要合并同类项，达到去重的效果。

然后以这个金字塔结构图为基础，再结合业务主体的泳道图，进行业务功能拆分。



3.2.4 关键指标

通过前面章节介绍的操作，已经把所有业务功能及业务主体都罗列出来，接下来就是根据下面的关键指标对业务进行标记，作为下一步拆分的依据。

- 公共的业务功能

被一个以上服务调用的接口方法都要独立出来，抽象成公共服务。

在微服务当中有两种类型的服务：一种是产品服务（Product Service）；一种是基础服务（Infrastructure Service）。

其中基础服务就是公共服务，在微服务架构当中如一些用户服务、计算服务、配置服务等都属于这种。

有些系统业务主体比较多，每个主体都会有很多规则，而且在每一个业务环节中都不一样，规则贯穿于整个流程，处处都有它的身影，和所有业务环节都有耦合。这时就要考虑把它抽取出来，做成一个统一的规则服务。

公共服务十分重要，相当于微服务架构的地基，公共服务独立出来以后，其他的服务直接调用公共服务即可，拆分也就更容易。否则每一个涉及的业务都要拆一遍，拆分公共服务相当于起到了瘦身的作用。

关键指数：★★★★★★★★★★

➢ 重点业务

此类业务偏向于业务主体，代表了未来业务发展的方向，更是高层领导十分关注的业务。

关键指数：★★★★★★★★★

➢ 对系统影响大的业务功能

对系统影响大的业务，包括一些对性能和资源消耗高的大户，如一些强计算业务及业务量十分大的业务。

关键指数：★★★★★★★★

➢ 经常变化的业务主体

有一些业务主体经常变化，并且在系统中与其他业务高度耦合，在每次变更时会涉及很多地方的更改，这时就要独立出来，与其他业务解耦，这样就可以避免频繁的改动对其他业务造成影响或埋下隐患。这一类业务拆分出去，相当于为系统排雷，提高系统安全系数。

关键指数：★★★★★★★

➢ 特殊业务主体

特殊业务主体，这种业务很独特，处理的逻辑与普通业务差别很大，处理流程不一样，这时如果把它放在系统当中处理，在业务逻辑上无法共用，与整个系

统显得格格不入，硬塞到系统里就是一个不伦不类的存在，就像一个土豆放到了一堆猕猴桃里。把特殊业务独立出来成为一个服务也是为了提高系统的稳定性，同时让代码有更好的可读性。

关键指数：★★★★★★

➢ 不同业务渠道

不同业务渠道的实现，就像现实中存在人工售票和自动售票这些不同的渠道一样，在很多系统中也存在这种情况，完成一个业务时有不同的实现方式，得到的结果是一样的，只是在处理时稍微有些差别而已。这种情况也要拆出来，这是为了保障系统的高可用性，有多种备选方案。

关键指数：★★★★★

➢ 业务复杂、带业务流程的

将复杂业务拆分成多个服务之后，复杂度大幅降低，让业务更加直观便于理解。

很多项目都有业务流程，按流程环节拆分成不同服务，每个环节相对独立，就像独立车间一样，然后可以针对某一业务环节向下拆分。

关键指数：★★★★

通过上面的关键指标识别出所有需要拆分的对象，接下来就可以根据项目实际情况按轻重缓急逐步来拆分。

3.3 粒度

在拆分过程中，粒度的选择也是一个关键要素。

有些人上来就问：微服务粒度应该是怎样的？

讨论粒度大小一定要结合具体的业务场景，与业务没有结合的任何讨论都是毫无意义的纸上谈兵。

有些人觉得既然可以通过微服务把业务做拆分，那么一不做二不休，干脆拆

得细一点，这种思想就有点矫枉过正了，人们经常陷入一种从极左到极右的困境，做事情不能太极端。

拆分的过程是循序渐进的过程，而不是一开始就全部拆成零部件，应该抓大放小，先粗后细。

拆得少，和没拆一样；拆得多，服务数量就会多，也就导致完成一个业务功能的组合因子特别多。而当组合因子太多时，就会出现问题难以定位的情况，同时也会带来很多问题，多服务的管理也是个难题，所以拆分粒度要根据自身能力来把握，也要考虑环境因素等其他客观条件，最终再来决定拆到什么粒度合适。

下面我们看两个关于粒度的例子，一正一反。



3.3.1 五笔字根

从第一版 86 版五笔输入法问世，到现在共经历三个版本，还有 98 版和新世纪版，但是一直以来使用最多的仍然是 86 版，说明 86 版最经典、最成功。

五笔字根每个都是独立的组件，从设计上，首先要保证所有的简体中文都可以通过五笔输入法打出来，就相当于做系统拆分时，业务范围要全部覆盖。通过提取字根成为组件，再将字根进行组合的方式来实现。

字根的粒度是关键，粒度太大无法组合，粒度太小就成了笔画输入法。

最终的结果就是先按基本笔画分成五个区，如图 3-1 所示。

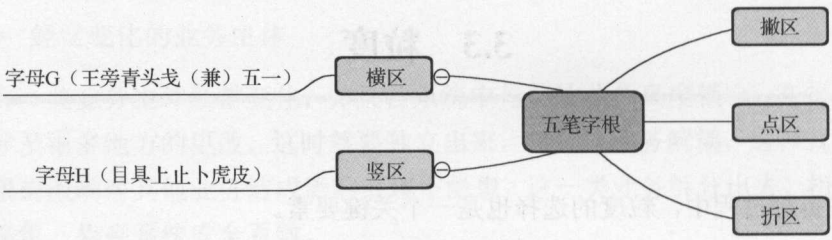


图 3-1　五笔字根图

然后每个区再进行细化，将字根分配到不同的字母键上，每个字母对应 3～7 个字根，如果粒度太小，一个字母上字根太多，既难记，又难以组合。

事实证明这种划分很合理，粒度不大不小，恰到好处。



3.3.2 被遗忘的字典师傅

电影《国产凌凌漆》里面有一段对白："我以为国家已经把我忘记了。"司令员说："怎么会呢？就算是一条内裤、一张卫生纸也有它的用处的。"

然而现实却是某教授接到编写小语种（普什图语）词典任务后，编写了 36 年被国家遗忘，这被称为文字史上的四大悲剧之一。

对于国家任务来说，正常是分配到大部委，直接分配到个人的时候粒度太小，而当任务拆解太细、太多时就会出现管理不善的问题。

在微服务架构当中服务不能划分得太细，因为粒度小必然导致服务数量多，服务数量多必然导致调用关系错综复杂，无论是服务的管理难度还是服务间的调用复杂度都大幅提升。最后可能得不偿失。



3.3.3 小结

粒度不是越细越好，一个微服务是同一种业务能力，是一系列功能的组合，而不是小到一个功能。

在粒度的选择上要达到某种平衡，在复杂度与灵活性之间寻求最佳方案。以解决实际问题为出发点避免为了拆而拆，拆的时候要考虑拆的意义，以及拆完之后维护的难度。

3.4 边　界

拆分的另一个关键要素就是定义好边界。边界定义要清楚，做到井水不犯河水。

如果两个国家边界不清可能会导致连年战争，民不聊生。村里农户之间边界不清，多一垄地、少一垄地可能会引发持续的争吵。服务之间如果边界定义不清就会埋雷，随时有返工的风险。

然而现实中很多事物的边界如此模糊，就像孤独与寂寞之间的边界，分也分不清楚。

边界的原则就是没争议，就像比赛裁判的判罚，不能引起双方争议。

➢ 边界清晰

就要像铁路警察一样，各管一段。

一个微服务只负责自己的一块业务，要和其他业务定义好明确的边界。边界越早定、越清楚越好。

有些时候因为边界不清，涉及多头管理会出现部门之间掐架的情况。

在微服务的划分里，如果业务划分不清，两个服务在功能上有重叠，同时由不同团队维护，那么将会出现两个团队都把业务往外推的情况，都说应该由对方来做，这种情况要杜绝。

➢ 分工明确

英格兰足球队中兰帕德和杰拉德无法共存的悲剧，就是因为分工不够明确。没有明确的分工，必然导致互相干扰、互为羁绊，从而发挥不出应有的水平。在微服务架构当中，如果也出现这种分工不明确的情况，那么系统维护起来将很麻烦。

3.5 本章小结

本章主要介绍了拆分的原则和方法、拆分粒度大小的平衡，以及边界的划分。

如何拆分取决于对架构的理解、对业务的熟悉程度；而拆成何种粒度更多取决于能够支撑的服务数量及随之带来的整体复杂度。

通过这一章的讲解，我想大家已经对如何拆分心里有数了，其实无关其技，唯手熟尔。

只要不是步子一下迈得太大，循序渐进慢慢拆、多论证总会得到理想的结果。



如何使用微服务

手握如此“利刃”如何才能最大程度发挥它的威力，这时就需要它的“使用说明书”。

本章主要讨论如何使用的问题，希望这本“使用说明书”能使您满意。

引入微服务一般有两种情况：新项目和老项目微服务化。

新项目在使用微服务时就像建新城，比老城改造的难度小得多，所以本章着重讨论老项目微服务化。

4.1 如 何 规 划

凡事预则立，不预则废，做好规划和准备工作是成功的开始。

好的规划对于系统来说非常重要，现在很多架构的问题是当初设计时缺少规划或者规划不足导致的，后期只能不断地打补丁，导致系统各种问题不断。为了避免这种情况，就要先做好规划工作。

软件架构在规划上要综合考虑很多方面，下面我们来和大良造学习一下如何规划。

《大秦帝国》中介绍了秦国新都咸阳城的规划，乙方项目经理和甲方接口人觉得按照现有的架构图施工，有些疑虑，需要和甲方高层再确认一下需求，遂前去拜见秦王。

当项目经理见到秦王和大良造时激动地说：“秦国变了，真的变了，我这种参与过十多个邦国都城改造的‘老司机’都从没有见过像秦国修建新都这样热血沸腾的施工场景，没见过这么大的阵仗。”

大良造静静地说：“秦国才刚刚上道，还没到发力的时候。”这阵势装的，我给满分。

项目经理吃惊地说：“啊！还有后劲？”潜台词是你这是要上天啊。

然后项目经理提出了自己的几个疑问。

（1）咸阳街道太宽，对于现在人口来说，会显得十分冷清。

（2）河里处处为商区，没有必要为山东商旅专设经商区域。

总的来说就是，项目经理觉得项目范围太大，新都的规划整体偏大，他觉得无论是街道的宽度还是经商的区域，甚至还有专门的办公地，都不太合常理，他参与过多个改造项目，都没遇到过这种情况，总之就是大处太多。

然而大良造仿佛看透了这一切，大良造说：“不能以现在的秦国人口来衡量日后咸阳城的大小，现在秦国的人口只有两百多万，只是一个栎阳城就够用了，但是十年以后，先夺河西和秦东，增加五六郡，再夺阴山与河套，国土再增加三四郡，人口将会达到七八百万，到时还会觉得咸阳城大吗？”

接着他又说：“秦国贫弱的根源之一就是商旅不振，要想使咸阳成为天下第一大都会，就必须扩张百工商贸，为山东商旅单独建立商区，通过筑巢来引凤。”

大良造说的每一句话都像一柄重锤敲击着项目经理的胸口，项目经理听到“十年”、“收复河西秦东”、“七八百万”的时候只想献上自己的膝盖。

我们从中可以得出以下几点启示。

（1）全局性

不谋万世者，不足谋一时；不谋全局者，不足谋一域。

规划都城的时候考虑根据未来的秦国发展情况，这里面又综合考虑了战略和经济上的规划。

秦国能统一六国创造不世之霸业，自身实力是一方面，眼光长远也是很

大的原因，当六国还在为了一郡一县而钩心斗角之时，秦国已经下了一盘很大的棋。

都城的规划和架构的规划是相通的，预测未来夺取河西与秦东，就像预见一个互联网产品未来用户数量增加一样，设计一个满足未来发展需要的都城就相当于设计一款可以满足未来发展的软件架构。

微服务架构的规划要像秦国都城一样从全局进行规划，包括业务和技术。业务方面要做好识别范围的工作，不能有遗漏；技术方面既要考虑技术架构的成熟度、稳定性，又要考虑将来如何升级，当用户数量和业务数量呈几何级增长之后系统是否可以正常支撑。在实现上也要考虑业务和技术的演进之路，以及技术要为业务服务的目标。

（2）针对性

大良造知道自身业务的痛点，改造时有的放矢，因为商旅不振，所以通过措施增强商旅，甚至为他们独立设区。这就像架构微服务一样，将主要的业务独立出来，变成一个独立的服务。

在知道如何解决问题之前，对未来的规划可能是错误的，这也是规划的难点。

（3）良性发展

从秦国都城的规划过程可以看出大良造是多么深谋远虑，如果没有建设一流的大都城，经济不可能发展这么快，也不可能有实力夺取河西秦东二地，进而实现人口的增长、经济的再次发展。整个过程像软件开发当中的迭代。他们是世界上最早采用迭代式开发的项目团队。

微服务架构设计时也要采用这种良性发展的方式，通过技术架构不断促进业务的发展。

（4）格局大

专业“老司机”参与过十几个邦国都城的改造、新建，但是没有一个像秦都这样的大手笔，从施工场景上就知道投入巨大。

咸阳城要做第一大都会，在规模上一定是当时最大的，从对话中就可以知道，这场轰轰烈烈的新建工作是史无前例的。从可扩展性的角度来说，这种规划是非常必要的，起码可以支撑百年的发展需要。

小结：

事后证明咸阳都城的规划非常成功，满足了秦国发展的需要，一直沿用一个多世纪，直到秦朝被灭。

这个过程之中，咸阳城一直作为秦国及至秦朝的首都而存在，始终很好地满足了国家发展的需要，没有因为人口的增加而导致堵车。要知道那个时候建好一座城之后是无法扩展的，不像现在城市可以扩充环路。没有出现因为规划不足导致重复投入、重复施工的问题，节省了资源。同时创造了巨大的价值，为日后统一六国奠定了坚实的基础。

微服务架构的规划就是要做到既能满足业务需求，又能促进业务的发展，同时始终为业务的发展保驾护航，最重要的是一次规划终身受用，不会出现架构无法满足业务发展需要的情况。如果达到这种效果，那么就算是非常成功的架构规划。

通过上面和大良造学习规划，我想大家已经对规划有了一个清楚的整体认识和足够的重视。

在微服务的规划上，分成业务和技术两部分，根据业务的现状和未来发展情况对技术架构进行规划。

4.2 微服务重构

前文讲了如何规划，接下来就该讲如何实现了。

它的实现过程本质上就是一次整体的重构，不同于传统的重构，通过调整程序代码改善软件的质量、性能，使其程序的设计模式和架构更趋合理，提高软件的扩展性和维护性。

微服务重构则是对整体架构进行大刀阔斧的改革，从而达到大卸八块的效果。

下面我们介绍一下重构时需要考虑的影响因素和重构的原则及方法。



4.2.1 影响因素

在使用微服务的时候要充分考虑环境因素，因地制宜，影响因素包括但不限于企业架构、战略、过程、技术、人。

环境因素影响非常大，如果不结合实际情况，只会是空中楼阁。

微服务最后做得是否成功，做到什么程度，很大程度上是由环境因素决定的，当环境因素十分不利，各种条件严重制约、阻碍时，微服务很难成功。

比如，它本来可以快速交付，但是受限于企业环境的基础设施，或者部署流程这些因素，就可能没办法实现快速交付，尽管它已经实现了快速开发，最后落地的周期仍然会很长。

所以要想能够用好微服务，充分发挥它的作用，那么首先应该改变的是企业自身的一些限制条件，如部署的审批流程、基础设施平台的搭建等。



4.2.2 重构原则

重构原则：就是要选择最合理、最经济、影响最小、安全系数最高的方式来达到预期效果。

➢ 前后端分离

若现有系统服务化，前后端需要先做分离，表现层和逻辑层做到解耦，这是做微服务改造的前提。

➢ 代价最小原则

马云说过，趁着晴天修房顶。在微服务改造过程当中，应该考虑选择一个合适的时机进行，如何把影响降到最低，当然是越早做越好，这样的代价最小。

如果有需求“冷冻期”那是绝佳的时机，然而现实中这种情况并不多见，通常都是一边使用，一边重构。

如果原系统继续上需求，继续增加功能又要做重构无异于抱薪救火，薪不尽，火不灭。

这时要尽可能将新需求在新的服务上面做开发，否则相当于要开发两遍，重构和测试验证的工作量十分巨大。

➢ 影响最小原则

在重构过程中，对系统的影响要降到最低，要始终保证系统能够正常使用。同时要通过试点的方式让风险提前暴露出来，及时改进，避免影响范围的扩大。

➢ 吐故纳新

微服务是一个契机，将原系统不合理的地方废弃，同时引入新的流程或规范对系统做优化，让业务更简单明了，流程更加优化，这样的重构才有价值。

如果业务没有任何变化，只是按照原来的重新做一遍，并不是最优的方案。最优的选择是从业务上做优化，然后再通过微服务去实现。



4.2.3 重构方法

首先要做的准备工作就是代码重构，要把新的技术框架定义好，搭建完毕。然后把原系统内部功能模块进行重构，归好类，每个模块内的代码要有高度相关性，或者同属于一个业务能力。接下来再使用下面的方法进行重构。

（1）接骨法

首先将原系统按照业务环节做拆分，拆分以后用“黑玉断续膏”再连接起来。这里的“黑玉断续膏”其实是服务化，Rest 接口或者是消息队列，即业务环节之间通过管道的方式进行先分再连。

拆后的每个服务都像乐高玩具组件一样，可以与别的组件进行自由组合。

（2）分批逃跑法

某军战争期间俘获了一个战俘，后来战俘得了一种病，左手烂掉了，他请求敌人把他的左手送回他的祖国，敌人很感动，就照做了。

不久战俘的右手也烂掉了，他也请求将右手送回祖国。

再后来他的左腿也被截肢，这一次当他要求把腿送回祖国时却遭到了拒绝。

他很不明白，问敌人为什么前两次可以而这次不行，敌人很郑重地说：“我们怀疑你在有计划地分期分批地逃跑。”

上面虽然是一个笑话，但是可以套用在微服务重构的情况上。

从老系统拆分业务出来，就是要一个一个地拆，分批地逃跑，最后用若干个新的应用代替原来的老系统。

（3）修路法

做老系统服务化时，既要成功微服务化，又要保证老系统正常使用。

这就像修路一样，修路时一般只占用部分车道，虽然有限流，但是车辆还可以部分通行。

修路时一般会封闭部分，做到分段封闭、分段施工。这样不影响正常使用，或者尽量降低影响。

微服务改造也应该这样来实践，将所有业务的上下游从直接调用改成 Rest 方式调用或者用消息队列的方式进行隔离，也就是修路里面的分段。

然后可以把入口处的业务或者选取某一环节分流出一部分试点业务用微服务实现，处理完之后再和老系统下一环节做拉通，老系统仍然正常使用，这样验证了试点业务之后，就可以把其他业务逐步迁移到微服务上。最后老系统的这个环节就可以废弃了，再开始下一业务环节的替换工作。

4.3 本章小结

本章主要介绍如何使用微服务，包含了微服务的规划、重构等内容。在对待微服务上，要警惕两种态度：一种是低估了微服务的复杂度；另一种是觉得微服务过于复杂。这两种态度都不可取，在微服务使用上应该怀着一颗敬畏之心，使用起来要小心谨慎，小心驶得万年船。

微服务架构是具有划时代意义的架构，与传统应用相比简直就是一股清流，极大满足了当前时代的需要，带来了诸多好处，正是生逢其时、前途一片光明，但是在使用上又非是任何人都能够驾驭好的，它就像一匹赤兔马，要有合适的环境、合适的主人才能发挥它的追风能力。

兵无常势水无常形，夫用兵者，贵在随机应变。如何使用要灵活，不能生搬硬套。



微服务的朋友圈

微服务并不是一个独立的存在，它有自己的朋友圈，也会和好友互动、点赞。

下面介绍一下它朋友圈的那些事儿。

5.1 好朋友容器

微服务的好朋友——容器，二者之间互动最多，甚至当容器还没有出现之前，微服务就一直在等待它的出现。

微服务再好，如果不能落地，那也不过是美好的空中楼阁，没有实际意义。

就像诸葛亮的火攻之计虽高，但是必须借助东风才可以发动攻势，没有东风就无法实施。

而容器就相当于微服务的东风，对微服务的发展起到极大的推动作用。

介绍完微服务，立刻就介绍容器，可见容器的重要性。容器的本质是基础设施，如果想做微服务，就要先把基础设施搭建好，为后续发展做好准备。

打好地基才能盖高楼，微服务要依赖于容器这个地基。



5.1.1 容器的来历

容器的标签就是隔离。

在容器技术出现以前，有一些虚拟化的技术，如 vmware，可以在物理机上划分若干个虚拟机，每个虚拟机都有自己的操作系统。

好处：通过创建虚拟机可以合理地利用资源，达到弹性伸缩的效果。

坏处：占用资源大，启动慢。

容器技术是在操作系统层之上的应用层进行了隔离。相当于虚拟化技术的一次升级。

容器可以将各种应用程序连同环境配置、依赖都打包在一起，就像刻了张光盘一样，可以在任何流行的 Linux 机器上运行，十分方便。

由于本身不包含操作系统，相比于传统的虚拟机，容器更轻量、更灵活，启动更快，可以像集装箱一样方便搬运。因为可移植，极大地简化了部署复杂度。



5.1.2 容器的好处

➢ 环境的一致性

很多时候生产环境的问题都是由部署引起的，很多时候部署的问题都是环境不一致引起的。这种情况几乎在每次部署上线时都会遇到。

比如，代码不一致、配置文件不一致、数据库脚本不一致，要么开发提交有遗漏，要么运维执行有误。

凡此种种都会引发各种问题，大大增加了部署的复杂度，让部署变成十分漫长痛苦的过程。很多项目的部署会持续一整夜，直到第二天清晨。

Docker 彻底解决了这一问题，当我们在开发环境使用 Docker 构建好应用的镜像后，可以直接在生产上使用同一个镜像，保证了线上、线下环境的一致性，从此世界和谐了。

➢ 减少了部署的重复工作

几乎所有的应用都对环境有配置依赖，就拿部署一个 war 包到 Linux 为例，需要搭建环境，安装 jdk、tomcat、mysql，配置一堆环境变量，这已经是比较简单的配置了。还有一些复杂的就不用说了。如果只是在一台机器上部署倒还好，但是在很多台机器上部署的时候就是“车祸现场”了。

现在用 Docker，我们只需要直接打包一个环境的镜像然后运行即可，无论

在多少台服务器上部署，都是一样的。

➢ 快速交付

部署变得容易的同时，可以实现快速交付，为持续集成创造了条件。

容器改变了传统的部署方式，开发人员只要将代码和配置提交到容器当中即可，技术人员不用像以往一样手动部署一堆应用程序、配置文件和数据库脚本。只需要动动小手，操作容器就可以，真是开发和运维的福音。

➢ 安全性高

容器和操作系统隔离，彼此互不干扰，无疑提高了系统安全性。

➢ 易集成

将不同应用部署在不同容器当中，解除了耦合，方便和其他应用集成。

➢ 自带体系

Docker 就是自带体系型球员，自带“小宇宙”，既是独立的，又是完整的小生态，可以被各处运行。

玩 HIFI 的人都知道，HIFI 并不是简单地购买一条好耳机或者一套昂贵的音响就万事大吉了，这只是一个开始，需要额外购置的配套设备很多，甚至需要专属的听音室，所以说 HIFI“毁三代”，如图 5-1 所示。

而容器最大的好处就是为你准备好了依赖环境以及所有的配套设备，它内部是一套完整的使用环境。可以拿来直接用，十分方便。

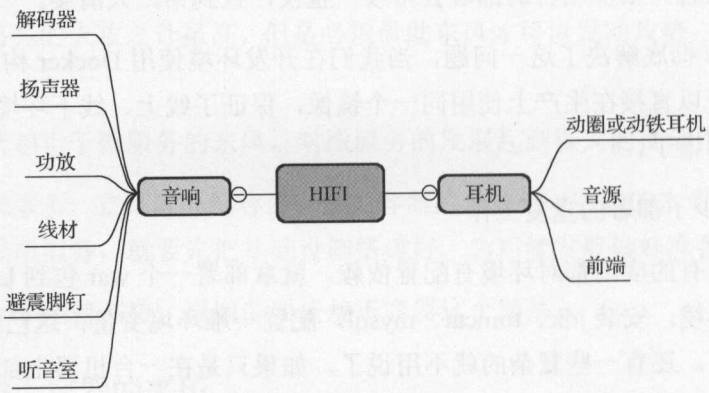


图 5-1　HIFI 配套指南



5.1.3 小结

微服务刚出现时面对的一个最大问题就是部署，因为服务拆成多个应用之后，所面对的部署复杂度大大增加了。这个时候传统的部署方式已经无法满足，需要新的部署方式来解放微服务的天性。

容器的出现恰恰解决了这个问题。

微服务和容器就是一对互相喂饼的队友。

因为微服务，容器更有价值。

因为容器，微服务才更能发挥出自身优势。

容器为微服务准备好环境，微服务就可以在上面自由发挥，容器搭好了戏台，微服务在上面唱大戏。而舞台不仅可以重复使用，还可以整体搬走，不需要每次重新搭建一个，从而节省了大量人力、物力和时间。

5.2 天生一对 DevOps

有微服务的地方就会看到 DevOps，两人几乎形影不离，吃住同行。



5.2.1 DevOps 是什么

DevOps 是开发+运维，更是一种能力，是为了打破开发与运维之间的鸿沟，将开发和运维的工作融合在一起，做到我中有你、你中有我，让两者不再是孤岛，而是统一的整体。

当开发和运维互相嵌入时，这就使开发延伸至生产中，生产反馈也加入到开发中，缩短了开发周期，有更高的部署频率，并且支持在不中断和不破坏当前服务的基础上，确保功能部署成功，同时也可以快速检测和修复缺陷。

DevOps 模式使整个产品周期的流程形成了一个闭环，不再是线性的流转。从计划到监控为一个迭代周期，滚动着进行下一个迭代周期的工作，如图 5-2 所示，几乎没有停滞，就像电影《雪国列车》里一直运行的火车。

虽然 DevOps 概念的提出没有多久，但很多人的 DevOps 经验应该比 DevOps 的年龄还要长。因为当团队规模很小时，从需求、开发、测试到运维的整个流程环节开发人员都会参与，开发不仅负责生产部署的工作，还要负责直接接受客户的反馈，并且立即进行行动、做出改进，这让整个项目的效率提升特别明显。这也是为什么我们在谈及 DevOps 这套方法论时常常会引出全栈式开发、敏捷等概念。

所以其实无形中我们已经采用了 DevOps 模式，只不过那时还没有这个概念，只知道这样做效率很高。

当时的人工干预过多，现在的 DevOps 更多借助自动化工具通过自动化的方式来进行构建，整个集成过程更为快速，通常是一提交代码就自动打包发布，相当于是一个 2.0 升级版。

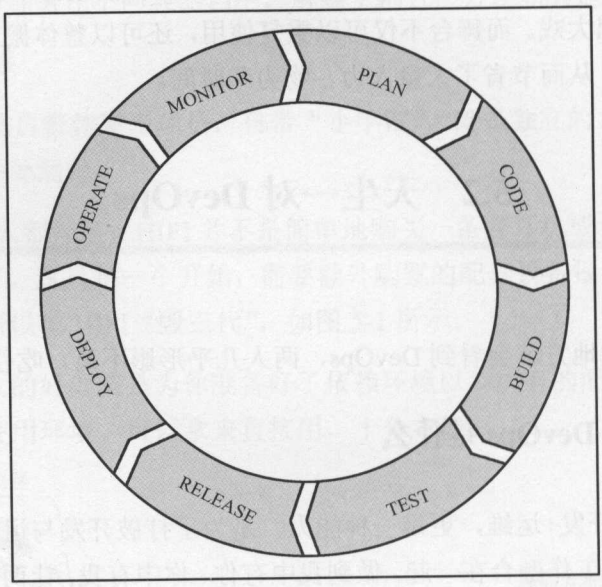


图 5-2　迭代周期



5.2.2　为什么采用 DevOps

随着项目发展，团队人数越来越多，并且成立了专门的运维团队来负责部署，这时我们与 DevOps 渐行渐远。同时发现职能分工明确带来的问题反而越来越多，经常会因为一些沟通原因，或者版本不一致等情况导致问题不断，开发与运维互相埋怨、推诿，这也导致交付流程效率偏低。

此时我们又十分怀念与 DevOps 一起的日子。同时发现我们貌似已经回不去了，因为人太多了，实现起来难度很大。

这时微服务出现了，借助微服务架构，团队拆分之后，又恢复到从前的小团队状态，这时又可以采用 DevOps 的方式。

通过 DevOps 很好地解决了上面的问题，让开发与运维变成同一战壕的队友，不再彼此独立甚至是对立的。

DevOps 是一种思维方式和习惯，更是一种战术思想，只有团队成员都用 DevOps 武装自己的头脑之后，才能发挥最大效果，否则可能只是形式上的 DevOps，而并不是真正的 DevOps。

当开发和运维之间壁垒森严时，开发就像一支足球队的前锋，运维人员像后卫，前锋只想着怎么进球，而后卫则考虑的是如何不丢球。由于双方有各自的打算，前锋喊：兄弟们给我冲啊。后卫则喊：兄弟们快回来啊，后面要守不住了。

而 DevOps 就相当于让前锋和后卫达成某种共识，大家的目标是统一的，就是为了赢球，赢球的前提是先守住，然后再进球。这样当球队处于防守时前锋也要负责骚扰对方后卫的出球，当进攻时后卫也积极参与，形成了全攻全守的打法。

DevOps 的亮点就是高效，微服务正好借助它来实现快速交付、快速反馈。

可以说微服务与 DevOps 之间的关系就是互相促进，共同进步的。



5.2.3 小结

DevOps 经常提到全栈式、敏捷等概念，微服务的先天优势是小、快、灵，二者之间的结合能发挥出最大的优势，所以说微服务与 DevOps 是天生一对。

5.3 SOA 是谁

微服务经常被人误认成 SOA。其实，这些人都是重度脸盲症患者。



5.3.1 接着忽悠

在某些领域所谓的“专家”比较多，因为这些领域有的是比较神秘的，如玄学；有的是人为的，如医学、SOA、云计算、大数据等。

SOA 曾经是重灾区，言必称架构如何如何，云里雾里，说的全是高（假）大上（空）的东西。有人掏了学费，买了教训，可谓一朝被蛇咬，从此是路人。再也不相信这些“专家”了，对这些热炒的词汇有了不同程度的抵触。

2005 年电影《无极》上映，导演陈凯歌说，十年之内没有人能看得懂这部电影。就像当年很多人都看不懂 SOA 一样。

十年之后已经被人遗忘，这也是我不想讨论 SOA 的原因，但是总有一些人在谈微服务时就要把 SOA 扯上，不知道 SOA“烦不烦”，微服务是“烦”了。



5.3.2 SOA 是个什么鬼

那就说一说 SOA 究竟是个什么鬼。

SOA 的概念是面向服务的架构，当时是一个很有前瞻性的理念。它本身描绘了一个很好的蓝图，通过它可以解决企业面对的一些问题。现在很多系统或多或少都会有 WebService 的功能，这都是面向服务的一种实现。

我曾经在“知乎”上看到一个专家解释 SOA 的帖子，耐心把几千字读完之后仍然没弄清它到底是干啥的，这种情况只存在于我读《周易》的时候。

也许是它太过抽象，太过宽泛，导致你说它是什么都行。所以当年 SOA 的很多“布道者”如今摇身一变，成了微服务专家，如转世灵童一般，说：你看，我当年没“忽悠”你吧，我们终于找到了 SOA 的最佳实践，就是微服务，微服务就这样被碰瓷儿了。

没办法，谁让微服务是面向服务的，名字中有这两个字，错不了。

其实面向服务说起来就和面向接口一样，都是软件架构中的普适原则。

当然有人会说那个什么“大神”就说微服务是 SOA。

尽信书不如无书，一定要有自己的判断标准。当说什么是什么的时候，这其实是个逻辑学的问题，是 is-a 的关系。

SOA 有几个重要标签，如 ESB、WS-，这些微服务全没有。下面来看二者的比较，如表 5-1 所示。

从外形上来看，SOA 一直在努力变成灵活的胖子，但灵活是一回事，胖是另一回事。

微服务看起来简直就是新版的葫芦娃兄弟，“萌”死个人儿。

表 5-1　微服务和 SOA

微服务	SOA
能拆分的就拆分	是整体的，服务能放一起的都放一起
纵向业务划分	是水平分多层
由单一组织负责	按层级划分不同部门的组织负责
细粒度	粗粒度
两句话可以解释明白	几百字只相当于 SOA 的目录
独立的子公司	类似大公司里面划分了一些业务单元（BU）
组件小	存在较复杂的组件
业务逻辑存在于每一个服务中	业务逻辑横跨多个业务领域
使用轻量级的通信方式，如 HTTP	企业服务产总线（ESB）充当了服务之间通信的角色



5.3.3　小结

微服务是不是 SOA 是一个仁者见仁，智者见智的问题，看从哪个角度解读，而问题本身并没有实际意义，应该把精力放在微服务上，如下一部分“技术实现篇”。

第二部分

技术实现篇

微服务的技术实现方式有很多种，可谓条条大路通罗马。

就像动漫《海贼王》里无论海贼从哪里出海，东海、西海、南海、北海，经历种种冒险之后最终都会汇集到拉夫特尔。

但是选择不同的路对应的结果不同，有的在半路遇上风浪就失败了；而有些虽然到达了，但是时间有先有后，如路飞团队经过佐乌岛之后就已经捷足先登快人一步了。

所以在技术选型上肯定要考虑效率的问题，同时又要考虑技术成熟度、性能、稳定性等因素。

这也是我选择 Spring Boot+Spring Cloud 这套微服务技术框架的原因。

因为这套框架使用简单、有强大社区支持、众多大牌加持、组件丰富且不断完善。

本部分主要围绕以 Spring Boot+Spring Cloud 框架为核心的技术实现进行讲解，并介绍其他一些在微服务使用过程中非常好用的辅助技术工具，如 Liquibase、Swagger、Sonar 等。



Spring Boot

Spring Boot 是 Pivotal 团队提供的全新框架，用于快速开发 Spring 应用。

Spring 为了让程序员解放天性而默默做了大量工作。

“我只能帮到这了!”——Spring Boot。

6.1 介　绍

Spring Boot 相当于模板，通过它可以快速创建应用，每个服务都可以通过 boot 引导创建，随意变形，通过定制可以变身为任意服务。

它的可复用性特别高，可以稍做配置上的修改就变成另一个服务，CV 大法的新传人。

即插即用的特性让它可以和其他框架无缝集成。

从功能属性上来说，Spring Boot 特别像是一款为马拉松选手准备的参赛包，让选手不用自己准备参赛用品，一切都已经准备妥当了，选手只需要把精力都放在比赛上就好。

相当于为软件框架提供了一站式服务，开发人员不用花太多精力在应用的准备、构建和部署环节上，只需要关注具体的业务实现即可。

6.2 特　点

- 快速构建 Spring 应用；
- 通过内嵌 tomcat、jetty 容器的方式，无须单独安装容器；
- 简化了 maven、gradle 配置；
- 通过注解自动化配置 Spring，实现零配置；
- 可以和其他主流框架如 Spring Cloud 无缝集成；
- jar 包直接运行。

6.3 Spring Boot 和微服务之间的关系

它们之间没有直接的关系。

有人说 Spring Boot 是微服务框架，这个说法是错误的，因为 Spring Boot 只是基础框架，并没有包括服务管理这些微服务的必要组件。

基础框架的意思是，使用它搭建的应用只是一个普通的 Spring 应用，或者是 Spring MVC 应用，只不过它要更轻量、灵活，只有当加入了其他服务管理方面的组件之后才叫微服务。

另一方面，微服务架构要求每一个应用都必须是小的、轻量的、灵活的，这就要求框架必须是轻量级的，而 Spring Boot 是目前 Spring 应用当中最轻量级的。

总结起来就是使用了 Spring Boot 的应用不一定是微服务；另一方面如果微服务没有采用 Spring Boot 的框架，那么又不是真正意义上的微服务，是假的微服务。

通过 Spring Boot 创造出来的应用相对于传统应用这样的凡夫俗子来说是“骨骼惊奇”的潜力股，这是它先天体质不同，但是如果没有借助其他外部工具如 Spring Cloud 打通任督二脉成为微服务，那么这辈子也就“平平淡淡才是真”了。

所以直接说它是“世外高人”，肯定是不对的，它只是具备这个潜质而已，以后究竟是走上“人生巅峰”或者“泯然众人”都不一定。

所以当听到有人说 Spring Boot 是微服务框架的时候，我也只能“呵呵”一笑。

6.4 如何使用

Spring Boot 是一个 Spring 的工具包，其特点是极大地简化了 Spring 应用的搭建、开发、部署过程。取消了 xml 文件的配置，只要通过注解的方式进行配置即可。

首先 Spring Boot 将系统必要的很多依赖打包在一起，以模块化的形式存在，在 maven 中只要引入 spring-boot-starter-parent，就相当于把这些依赖同时都引入了，不用再去手动创建和管理这些依赖。想要开启哪个功能，就在下面增加 dependency，比如这里只需要加入 spring-boot-starter-web 就打开了 Web 容器的开关，就可以直接运行应用，不需要另外安装配置 tomcat。

通过 Spring Boot 开发 RESTful Web 服务的方式非常简化，只需要通过几行代码就可以发布一个服务。

第一步：pom.xml 配置如下。

```
<parent>
        <artifactId>spring-boot-starter-parent</artifactId>
        <groupId>org.springframework.boot</groupId>
        <version>1.3.8.RELEASE</version>
        <relativePath />
    </parent>
    <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
<build>
        <plugins>
            <plugin>
```

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
                </plugin>
            </plugins>
</build>
```

第二步：编写代码。

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
@SpringBootApplication
@RestController
public class RestServiceApplication {
      public static void main(String[] args) {
            new SpringApplicationBuilder(RestServiceApplication.class).web(true).run(args);
      }

      @RequestMapping(value = "/hello" ,method = RequestMethod.GET)
      public ResponseEntity<String> hello() {
          System.out.println("hello rest service");
          return new ResponseEntity<>("hello rest service", HttpStatus.OK);
      }
}
```

第三步：执行打包命令。

在第一步 POM 已经增加了 maven 的插件，然后可以通过 mvn package 命令将应用生成可以直接运行的 jar 包，在 target 目录下生成 rest-service-1.0.0.jar。

```
C:\xxxcode\rest-service>mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building rest-service 1.0.0
[INFO] ------------------------------------------------------------------------
```

```
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ rest-service ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ rest-service ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to C:\xxxcode\rest-service\target\classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ rest-service ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\xxxcode\rest-service\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ rest-service ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to C:\xxxcode\rest-service\target\test-classes
[INFO]
[INFO] --- maven-surefire-plugin:2.18.1:test (default-test) @ rest-service ---
[INFO]
[INFO] --- maven-jar-plugin:2.5:jar (default-jar) @ rest-service ---
[INFO] Building jar: C:\xxxcode\rest-service\target\rest-service-1.0.0.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.3.8.RELEASE:repackage (default) @ rest-service ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 14.589 s
[INFO] Finished at: 2017-02-20T15:42:32+08:00
[INFO] Final Memory: 28M/194M
[INFO] ------------------------------------------------------------------------
```

第四步：执行运行命令。

```
C:\xxxcode\rest-service>java -jar target/rest-service-1.0.0.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
```

```
  \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
   '  |____| .__|_| |_|_| |_\__, | / / / /
  =========|_|==============|___/=/_/_/_/
  :: Spring Boot ::        (v1.3.8.RELEASE)
2017-02-20 15:45:57.909  INFO 12112 --- [                 main]
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/docs/],produces=[text/html]}" onto
public java.lang.String org.springframework.boot.actuate.endpoint.mvc.DocsMvcEndpoint.browse()
2017-02-20 15:45:58.182  INFO 12112 --- [                 main]
o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
2017-02-20 15:45:58.220  INFO 12112 --- [                 main]
o.s.c.support.DefaultLifecycleProcessor  : Starting beans in phase 0
2017-02-20 15:45:58.647  INFO 12112 --- [                 main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 1234 (http)
2017-02-20 15:45:58.665  INFO 12112 --- [                 main]
com.jzb.RestServiceApplication           : Started RestServiceApplication in 13.31 seconds
(JVM running for 15.04)
```

这样一个工程就启动了，通过浏览器打开 http://localhost:1234/hello，如图 6-1 所示。

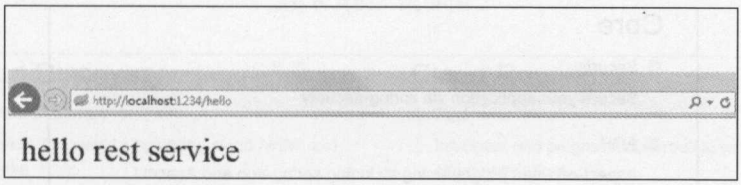


图 6-1　hello rest service 图

6.5　可定制化

官网提供了可定制化功能，如图 6-2 所示，地址是http://start.spring.io/。

可以在界面上定制工程，按自己的需要选择 Maven 或 Gradle 进行构建，添加组件的依赖，就可以直接生成工程包，非常方便。甚至一些 IDE 里面也已经集成了这个可订制化工具。

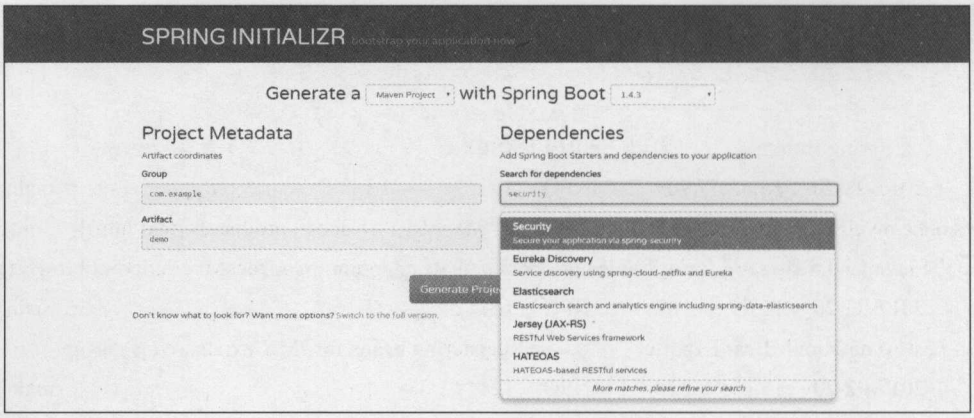


图 6-2　定制界面图

整个过程就像攒机一样，完全按照自己的需求选择组件，进行定制，非常灵活方便。

Spring Boot 就是一个机箱，里面集成不同的 CPU 和硬盘等组件，下面只列举了其中提供的一部分组件，如图 6-3(a)、图 6-3(b)、图 6-3(c)、图 6-3(d)所示。

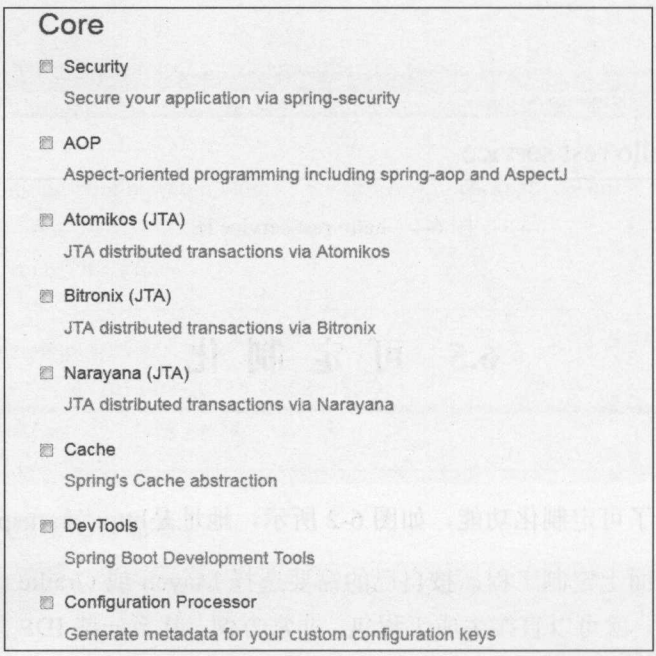


图 6-3(a)　组件图 1

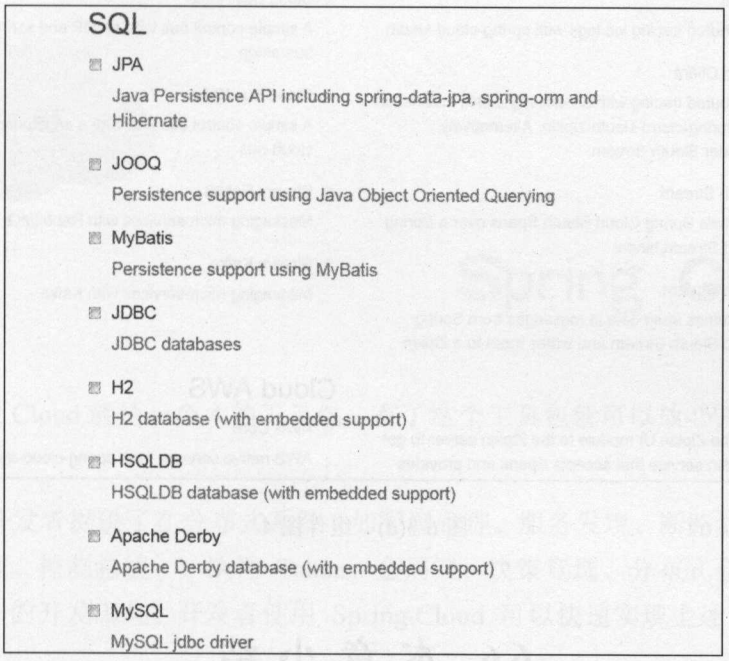


图 6-3(b)　组件图 2

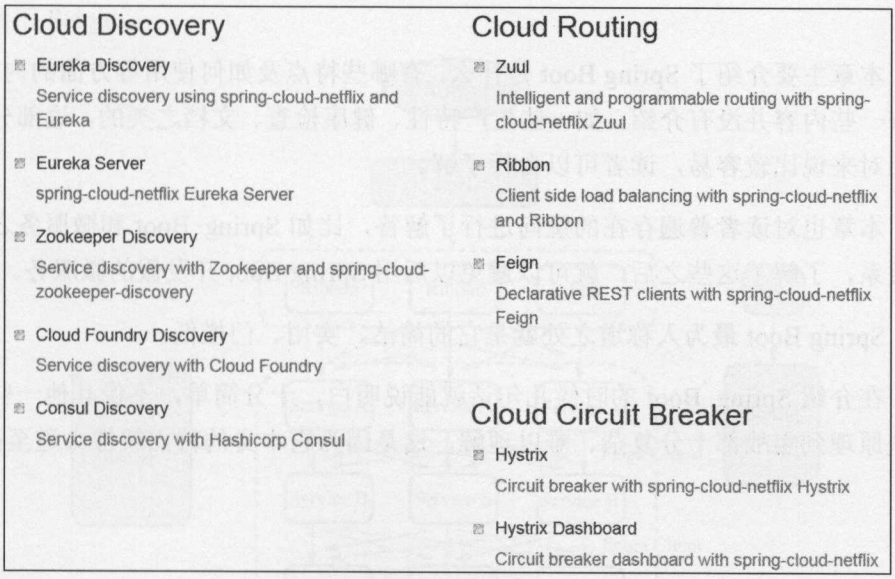


图 6-3(c)　组件图 3

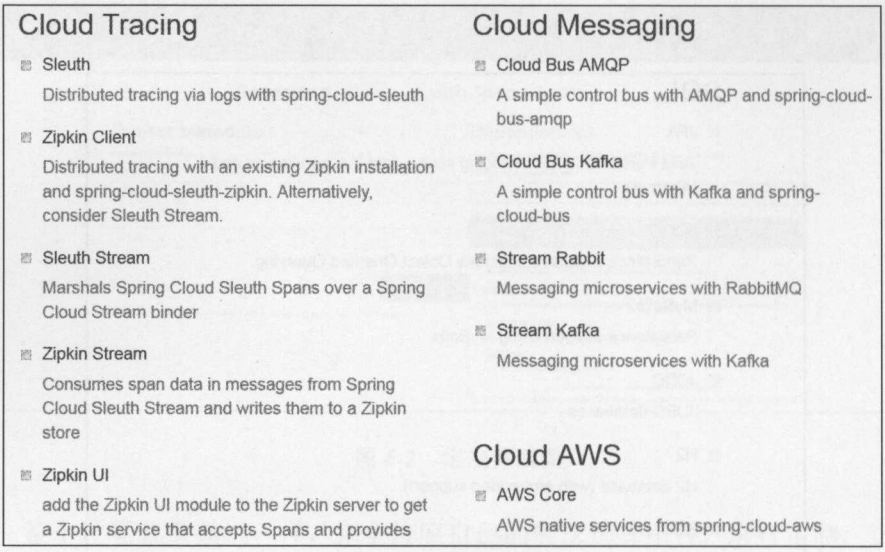


图 6-3(d) 组件图 4

6.6 本章小结

本章主要介绍了 Spring Boot 是什么，有哪些特点及如何使用等方面的内容。还有一些内容并没有介绍，如一些生产特性、健康检查、文档之类的，这部分内容相对来说比较容易，读者可以自行了解。

本章也对读者普遍存在的疑问进行了解答，比如 Spring Boot 和微服务之间的关系，了解了这些之后，就可以避免以后用 Spring Boot 开发假的微服务。

Spring Boot 最为人称道之处就是它的简洁、实用、门槛低。

在介绍 Spring Boot 的时候几句话就能说明白，十分简单，不像其他一些技术从原理到实战都十分复杂、难以理解，这是因为它本身的特点就像大道至简。

第 7 章

Spring Cloud

Spring Cloud 就是一个大的工具包，有了这个工具包就可以放心开发微服务应用了。

它为开发者提供了在分布式系统（如配置管理、服务发现、断路器、智能路由、微代理、控制总线、一次性 Token、全局锁、决策竞选、分布式会话和集群状态）操作的开发工具。开发者使用 Spring Cloud 可以快速实现上述这些功能。

通过 Spring Cloud 组件搭建的系统架构如图 7-1 所示。这其中涉及注册中心 Eureka、智能路由 Zuul、负载均衡 Ribbon、配置中心 Spring Cloud Config 及服务间调用 Feign。

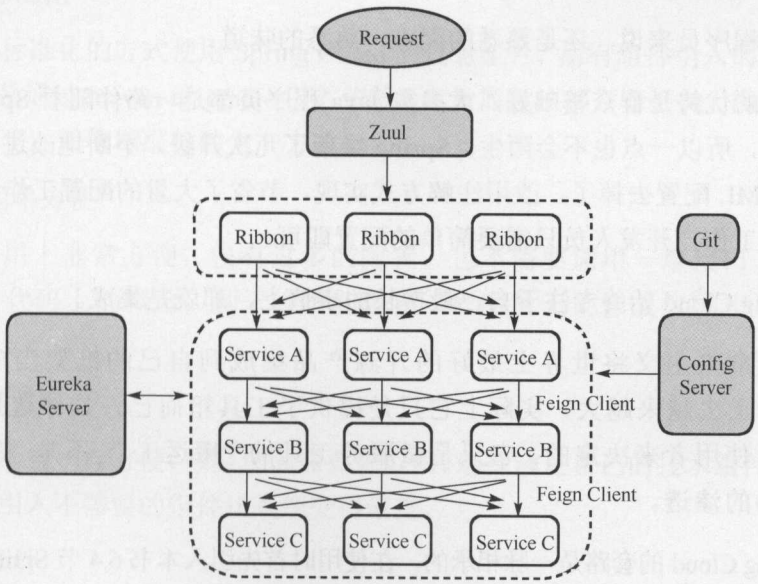


图 7-1　系统架构图

通过这些组件就可以打造出了一个比较完整的微服务架构，本部分主要围绕这些组件进行讲解。

服务消费者的请求执行过程如图 7-2 所示。无论是通过浏览器还是其他渠道过来的请求首先会通过代理服务器也就是网关经过鉴权等操作进入，并根据请求路径进行路由，然后通过负载均衡器进行分发，负载均衡器会从注册中心读取服务列表，根据服务列表进行选择，最终请求到目标服务。

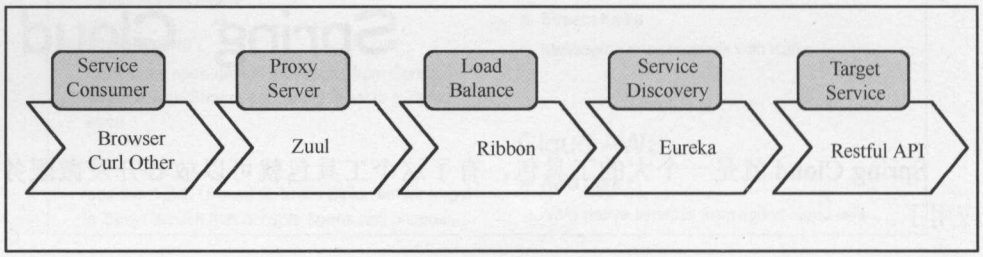


图 7-2　执行过程图

7.1　Spring Cloud 优势

对于程序员来说，还是熟悉的配方、熟悉的味道。

最大的优势是群众基础好，大多数 Java 程序员都是一路伴随着 Spring 的成长而成长，所以一点也不会陌生。Spring 经历了几次升级，不断地改进，把曾经大量的 XML 配置去掉了，改用注解方式实现，节省了大量的配置工作量和不必要的调试工作，开发人员只需要简单的配置即可。

Spring Cloud 始终专注于自己最擅长的事情上，那就是集成。

通过拿来主义将世界上最好的开源产品集成到自己的框架当中，虽然 Spring 看上去越来越大，实际上它只是提供了工具箱而已，具体选择哪个工具都是由使用者来决定的。它只是微服务工具的“搬运工”，不是“生产者”，没有代码的渗透。

Spring Cloud 的套路是一脉相承的，在使用时首先引入本书 6.4 节 Spring Boot 的依赖，然后添加 Spring Cloud 的依赖，最后根据需要打开对应组件的开关就可以了。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR7</version>
                <type>pom</type>
                <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

7.2 Spring Cloud 特点

➢ 功能齐全

涵盖了微服务架构的各个方面，不一而足，充分考虑了微服务使用者的需求，集成了很多公司的优秀的开源框架。

➢ 标准化

通过标准化的方式使用 Spring Cloud 的任意组件，所有组件引入的方式是通用的，都是通过 maven 或 gradle 引入依赖，然后通过注解打开开关，简单加个配置就可以了，就像标准化的生产方式。

➢ 简单方便

在使用上非常方便，没有过多的配置，也不需要调用一堆接口，与组件之间没有代码上的耦合，利于维护和替换。程序员无论修改代码还是新增功能都简单。

➢ 按需取用

像自助餐一样方便，按自己的需要来取用，选择满足自己的技术组件就可以，不会因为引入不需要的组件让系统变得笨重。

➢ 轻量

虽然 Spring Cloud 包含了许多功能组件，看似是非常大的框架，但是实际使

用时会发现它的每个组件都是非常轻量的。

➢ 易扩展、易维护

在组件的选择上可以像乐高积木一样不断地增加组件，拼成一个更大的玩具。比如，在原有架构基础之上增加负载均衡的组件，直接增加就可以，再想增加路由时仍然是直接增加不需要改原来的代码。同时又可以很方便地去掉不需要的组件，不会像其他第三方组件那样出现“请佛容易送佛难”的情况。

➢ 可复用性

组件之间的可复用性很高，简单修改一下，一个服务就可以变身成另一个服务，不用完全重新开发，节省了很多工作量。

7.3 Spring Cloud 之服务管理

传统应用的弊端早就有共识，架构师也一直在寻找突破这种架构局限性的方法，通过类似活字印刷的方式，实现组件之间的解耦。

不过不同于活字印刷当中组件的简单易管理，软件架构中要想实现全部拆散成组件的方式，随之而来的管理难度会放大很多倍，这时需要相应框架来进行服务管理。

在微服务架构当中服务治理是一个出现频率很高的词汇，但是我对“治理”一词一直有点疑惑，它是从英文 governance 翻译过来的，我对这个翻译一直不敢苟同，我觉得翻译成服务管理就很好。

“治理”容易听成“治你”，所以就变成服务“治你”了，服务本来是为你服务的，所以我觉得还是叫“服务管理”比较容易理解。

服务管理就是管理服务的，是为了能让服务正常运行，不出问题，实现高可用性。就像小到社区有物业进行管理，大到国家有各种职能部门来管理。

管理只是一方面，服务是另一方面，比如在我们很多政府机构的墙上都有“为人民服务”几个大字。

所以服务管理是为了管理这些服务，更是为服务服务。

服务管理相当于为微服务引入了一套英式管家服务体系，从三口之家到四世同堂，当人口多了之后要有专人来进行日常管理。而管家服务做得好不好的标准就是让你既享受了管家服务，又感受不到管家的存在。Spring Cloud 正是这样的“管家”。你不用主动找管家，一切自然有人替你打理，也不会在用餐或休息时被管家打扰，这样的管家就是称职的。

7.4 注册中心 Eureka

微服务架构中一个最重要的部分就是服务管理，而服务管理就是通过注册中心来实现的，注册中心可以说是最核心的一个组件，以至于离了哪个组件都可以，但是唯独不能缺少注册中心。有没有注册中心甚至可以作为评判一个架构到底是不是微服务的标准。



7.4.1 什么是注册中心

说注册中心之前我们先说一下什么是“组织”，“组织”是“组”和“织”两个字，“组”代表或三或五或更多成员，“织”代表成员之间的一种网状关系，是交织在一起的。

组织的重要性在于可以把或松散或紧凑的独立个体变成一个大集合，有了边界和范围。

而注册中心就是一种组织，也叫发现注册中心，服务端与客户端之间是发现与被发现的关系、注册与被注册的关系。

在分布式系统中，注册中心负责将所有的服务进行统一管理，把所有服务变成一个有机的整体，不再是独立的个体。通过组织对外提供集中式服务，而不再是由每一个个体单独提供服务。



7.4.2 为什么要有注册中心

微服务架构中为什么要有注册中心？

传统应用只有一个服务，当代码调用 Rest API 的时候，通常需要知道服务实

例的网络地址，如通过 IP+端口的形式，代码可以从一个配置文件中读取。

但是微服务里面的服务实例不止一个，可能成百上千个，而且微服务的服务实例的网络地址是动态分配的。由于自动扩展、注销、变更，服务实例的地址也经常变化。

如果按照以前的方式根本玩不转，这个时候怎么办？

因此整个系统需要一套服务发现机制，通过服务发现之后，相当于找到了组织，所有的服务组成了一个聚合体，再调用某个服务的时候可以通过服务名的方式进行调用，而不用关心具体的 IP 和端口。

注册中心的另一个好处是，便于被外界发现和使用，如果没有注册中心，有的服务可能被遗忘或者压根就没有人知道，而通过注册中心之后，所有的服务像和尚头上的虱子一清二楚。

综上所述，在微服务的分布式系统中，注册中心的意义特别重大，能不能做好服务管理工作，决定了一个分布式架构的成败。



7.4.3 Eureka 来历

Eureka 是 Netflix 开发的服务发现框架，Spring Cloud 将它集成在自己的子项目 spring-cloud-netflix 中，实现 Spring Cloud 的服务发现功能，如图 7-3 所示。无论应用服务还是应用客户端都注册到服务中心上，然后应用客户端再通过 Rest 方式请求应用服务。

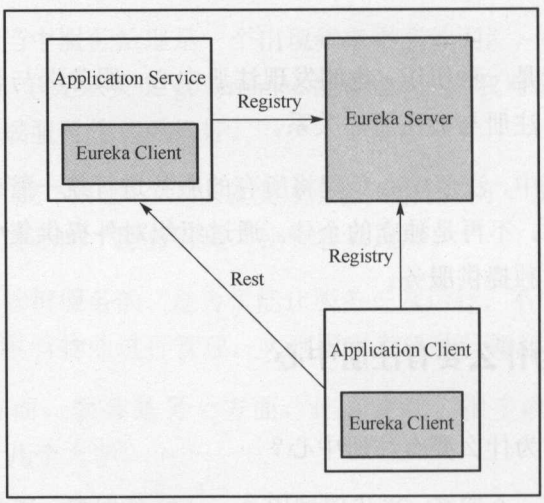


图 7-3　Eureka 服务发现架构图

Eureka 作为注册中心承载了所有服务的注册与发现功能，所有注册到它上面的服务都是 Eureka 客户端，客户端之间再调用的时候就可以直接通过服务名进行调用。



7.4.4 Eureka 机制

Eureka Server 是服务注册中心，提供服务注册功能，采用的客户端发现模式。

每一个服务实例被启动时，根据配置，它会注册到 Server 上，同时它的网络地址会被写到注册表上，这样 Eureka Server 中就有了所有服务实例的信息。

在这个过程中 Server 端暴露出一个地址，然后所有的 client 都向这里注册，就好比在《水浒传》当中，梁山作为根据地大本营发挥着注册中心的作用。

梁山既可以服务端发现，又可以客户端发现。这两种发现模式的区别在于，前者是梁山主动发现其他服务，如卢俊义、徐宁等。后者是则大多数投奔梁山的人所使用的方式——逼上梁山。

梁山暴露出一个地址，各路人物知道这个地址就来投奔（注册）了，刻了字的脸就相当于通行证，当然也要是三十六天罡、七十二地煞才行。

梁山好汉就通过这两种方式，成功注册成了 108 个服务，梁山本身也是一个服务，是注册服务。

当服务端发现时，服务端要包含业务逻辑，这时它是中心化的，比较而言，客户端发现更灵活，是去中心化的，轻量、耦合更少。

有了服务发现之后，就可以统一向外部提供服务，如替朝廷打完大辽、打方腊，最后通过缩容、服务降级等策略把这些服务都干掉了。

同时 Eureka 提供了仪表盘功能，通过 Eureka 的监控页面，可以在页面中直观地看到所有服务的状况。

为了保证服务的可用性，通过 Eureka 心跳机制，当某个节点服务在规定时间内没有发送心跳信号时，或者当服务实例终止时，Eureka 会从服务注册表中把这个服务节点移除，这个服务实例的注册表通过心跳机制动态刷新。

Eureka 还提供了客户端缓存的机制，即使所有的 Eureka Server 都“挂掉”，客户端仍可以利用缓存中的信息调用服务节点的服务。Eureka 通过心跳检测、健康检查、客户端缓存等机制，保证了系统具有高可用性和灵活性。



7.4.5 如何使用

第一步：创建注册中心服务端工程 eureka-server。

（1）pom.xml 增加依赖。

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.8.RELEASE</version>
    <relativePath/>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka-server</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR7</version>
```

```
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
```

application. yml 增加：

```
server:
  port: 8761
eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false
    fetchRegistry: false
  serviceUrl:
    defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

配置了端口 8761，所有服务实例向这个地址进行注册。因为这个是注册中心的 Server，它不需要向自己注册，所以 registerWithEureka 和 fetchRegistry 都设成 false。

（2）服务端 Java 代码。

```
@SpringBootApplication
@EnableEurekaServer
public class ServiceRegistryApplication {
    public static void main(String[] args) throws Exception {
        SpringApplication.run(ServiceRegistryApplication.class, args);
    }
}
```

其中：@EnableEurekaServer 注释声明这个类是 Eureka 的 Server。

服务启动之后通过浏览器访问http://localhost:8761/，可以看到注册中心的 Server 已经启动成功，如图 7-4 所示。

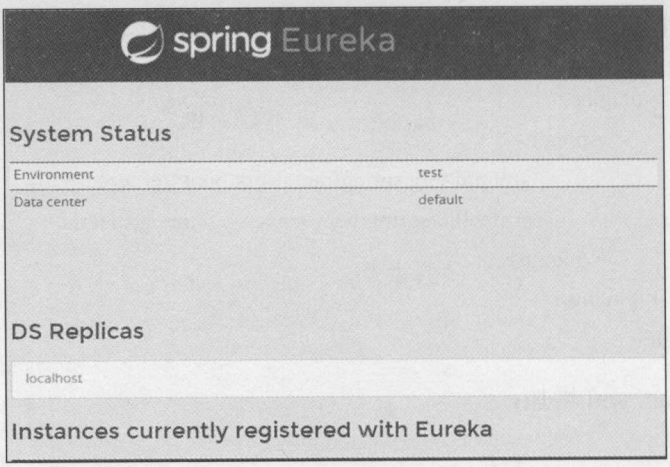


图 7-4　仪表盘界面

这是 Eureka 自带的监控界面，相当于电影院里滚动显示的大屏。电影院相当于一个服务发现中心，每一个场次的电影都是一个服务，观众是服务调用方来消费这些服务的。电影院的大显示屏上会滚动显示当前可供观看的电影场次，上面显示的电影就是可供正常调用的服务，上面没有的影片就是已经下线的服务。

我们可以很直观地看出服务的状态，哪些服务可用，哪些不可用。

第二步：注册中心客户端工程 eureka-client。

Eureka 客户端进行服务实例注册和注销，Spring Cloud 简化了 Eureka 的服务实例自动注册。本书后面所有服务都需要打开 Eureka 的开关。

步骤如下。

（1）pom. xml 配置如下。

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.8.RELEASE</version>
</parent>
```

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Brixton.SR7</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
```

（2）代码增加一行注解@EnableDiscoveryClient。

通过@EnableDiscoveryClient 打开开关，这个类就可以被注册中心发现。

```
@EnableDiscoveryClient
@SpringBootApplication
public class EurekaClientApplication {
    public static void main(String[] args) throws Exception {
        SpringApplication.run(EurekaClientApplication.class, args);
    }
}
```

（3）application. yml 增加。

```
#Eureka Server Configuration for DEV
eureka:
  client:
    registerWithEureka: true
    fetchRegistry: true
    availabilityZones:
      default: ${APPLICATION_DOMAIN:${DOMAIN:defaultZone}}
    serviceUrl:
      defaultZone: ${vcap.services.sessionqaeureka.credentials.uri:http://localhost:8761/eureka/}
#Eureka Server Configuration for DEV
eureka:
  instance:
    leaseRenewalIntervalInSeconds: 50
    leaseExpirationDurationInSeconds: 30
    preferIpAddress: false
    statusPageUrlPath: /eurekaclientservice/admin/info
    healthCheckUrlPath: /eurekaclientservice/admin/health
```

leaseRenewalIntervalInSeconds、leaseExpirationDurationInSeconds 这两个是自己设置的轮询时间间隔。上面的 50 和 30 是作者在实际项目当中的设值，如果不设置系统有默认值。

leaseRenewalIntervalInSecondsDefault 解释：

客户端每隔多少秒发送心跳给服务端，表明客户端还活着，默认 30 秒。

leaseExpirationDurationInSeconds 解释：

相当于拳击比赛当中，裁判对拳手倒数十秒，如果拳手不能给个回应，那么裁判就会判失利，类似的注册中心如果在规定时间内没有收到服务的心跳，那么注册中心就会判这个服务失利，让这个服务出局。默认 90 秒。

详细说明如下。

leaseRenewalIntervalInSecondsDefault:30.

Indicates how often (in seconds) the eureka client needs to send heartbeats to eureka server to indicate that it is still alive. If the heartbeats are not received for the period specified in leaseExpirationDurationInSeconds, eureka server will remove the instance from its view, there by disallowing traffic to this instance. Note that the instance could still not take traffic if it implements HealthCheckCallback and then

decides to make itself unavailable.

leaseExpirationDurationInSecondsDefault:90.

Indicates the time in seconds that the eureka server waits since it received the last heartbeat before it can remove this instance from its view and there by disallowing traffic to this instance. Setting this value too long could mean that the traffic could be routed to the instance even though the instance is not alive. Setting this value too small could mean, the instance may be taken out of traffic because of temporary network glitches.This value to be set to atleast higher than the value specified in leaseRenewalIntervalInSeconds.

这时再启动 eureka-client，启动成功后，刷新 eureka-server 仪表盘，就可以看到服务注册到上面了，如图 7-5 所示。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
EUREKA-CLIENT	n/a (1)	(1)	UP (1) - 192.168.254.86:eureka-client:8763

图 7-5　仪表盘图

这样服务就成功注册了，注册之后的服务就可以被其他服务直接调用。

7.5　网关 Zuul

古语有：一夫当关，万夫莫开。

网关 Zuul 就在系统里起到同样的作用，它是系统的门户、城市的关隘、公园的检票口。

服务网关 API Gateway 可以有很多实现方法，如 Nginx、Zuul，甚至是一个 Node.js 的服务端。它们最重要的作用是为前台提供后台服务的聚合，提供一个统一的服务出口，解除它们之间的耦合，同时负责鉴权、认证、安全和跳转等作用。



7.5.1 Zuul

Zuul 原为 Netflix 公司开发，后被招安到 Spring Cloud 框架当中。

Zuul 是边缘服务，用来提供动态路由、监控、授权、安全、调度等功能，将权限控制等一些业务逻辑抽离出来，单独放到 Zuul 里，使得服务组件更简单，具有更好的可复用性。

Zuul 还负责将请求路由到服务上，起到了前台接待人员的作用，负责访客接待，当然也可以不使用 Zuul，这时请求直接与后台的服务交互，但是这样破坏了软件架构中的面向接口原则，相当于不要前台接待人员了，这时所有访客都直接进公司里面自己找人，这种情况就会比较混乱。

Zuul 执行过程如图 7-6 所示。

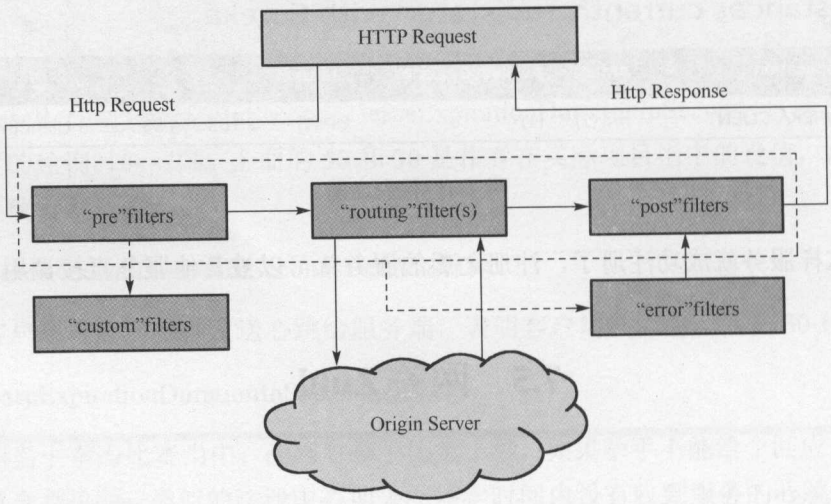


图 7-6　Zuul 执行过程



7.5.2 反向代理

Zuul 是反向代理工具，类似的工具还有 Nginx。

代理分为正向代理和反向代理，我们来了解一下。

有一些事情是你不方便做的，比如你通过代购买东西，这时代购就是一个代理人的角色，卖方不知道是你在买。这些情况都是正向代理。

反向代理的例子比如你打 10086，你不知道接电话的是谁，你只知道打这个电话就有人帮你解决问题就可以了，至于他们是如何分配，分配了哪个坐席给你，你是不用关心的。

而在应用系统中反向代理类似于 10086 这种情况，就是为后端的多台服务器提供负载平衡，或为后端较慢的服务器提供缓冲服务。

综上所述：

正向代理代理的是客户端，反向代理代理的是服务端；

正向代理，服务端不知道客户端是谁；

反向代理，客户端不知道服务端是谁。

Zuul 在这里是反向代理，代理了后台的所有服务端，前台请求不需要知道真正的服务端是谁，只要交给 Zuul 就可以了，Zuul 负责路由到正确的服务上。



7.5.3 如何使用

首先需要一个 eureka server 用于服务发现，然后需要两个不同服务来验证 zuul 的路由功能。两个服务分别是 order service 和 product service，还需要路由 zuul service。

第一步：创建 order-service 工程。

工程结构如图 7-7 所示。

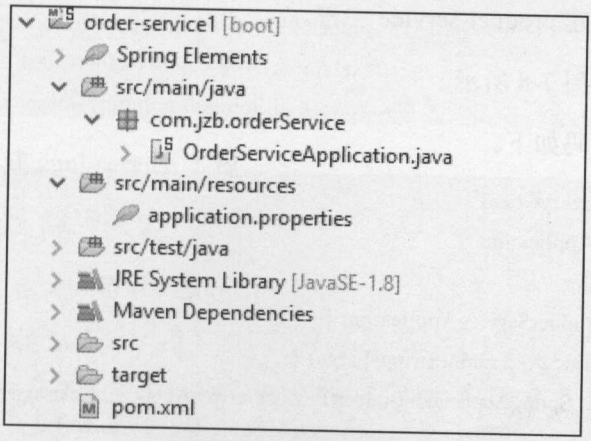


图 7-7　order-service 目录结构图

（1）程序代码如下。

```
@EnableDiscoveryClient
@SpringBootApplication
@RestController
public class OrderServiceApplication {
    public static void main(String[] args) {
        new SpringApplicationBuilder(OrderServiceApplication.class).web(true).run(args);
    }

    @RequestMapping(value = "/hello" ,method = RequestMethod.GET)
    public ResponseEntity<String> hello() {
      return new ResponseEntity<>("hello order service1", HttpStatus.OK);
    }
}
```

（2）application.properties 配置如下。

```
spring.application.name=order-service
server.port=3331
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/
eureka.instance.lease-renewal-interval-in-seconds=50
eureka.instance.lease-expiration-duration-in-seconds=30
```

其中 spring.application.name 配置的是服务 ID，对应在 Eureka 监控界面看到的服务名称，相当于服务的别名，被调用时会被引用，如在 Zuul 中配置路由时就会用到这个名称。

第二步：创建 product-service 工程。

工程结构如图 7-8 所示。

（1）程序代码如下。

```
@EnableDiscoveryClient
@SpringBootApplication
@RestController
public class ProductServiceApplication {
    public static void main(String[] args) {
        new SpringApplicationBuilder(ProductServiceApplication.class).web(true).run(args);
    }
```

```
    @RequestMapping(value = "/hello" ,method = RequestMethod.GET)
    public ResponseEntity<String> hello() {
      return new ResponseEntity<>("hello product service1", HttpStatus.OK);
    }
}
```

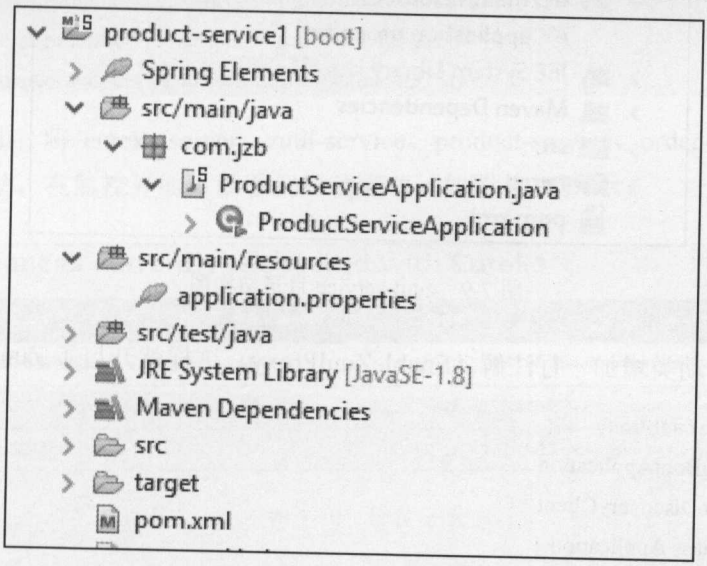


图 7-8　product-service 目录结构图

（2）application.properties 配置如下。

```
spring.application.name=product-service
server.port=2221
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/
eureka.instance.lease-renewal-interval-in-seconds=50
eureka.instance.lease-expiration-duration-in-seconds=30
```

第三步：创建 zuul-service 工程。

目录结构如图 7-9 所示。

（1）工程增加 Zuul 依赖，pom.xml 增加。

```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-zuul</artifactId>
        </dependency>
```

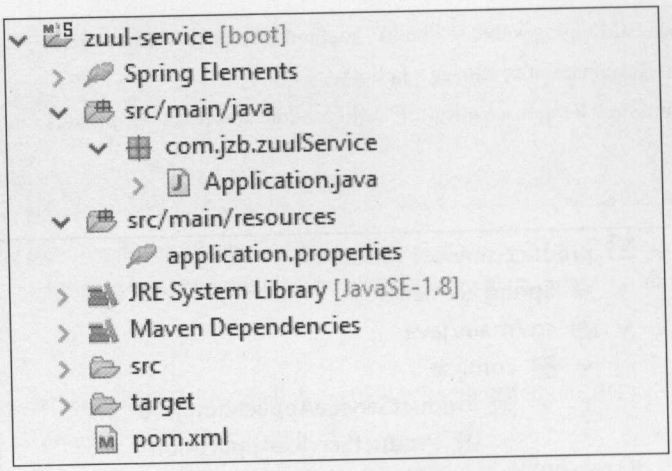


图 7-9　zuul-service 目录结构图

（2）启动类增加一行注解@EnableZuulProxy，这样就开启了 Zuul 的功能。

```
@EnableZuulProxy
@SpringBootApplication
@EnableDiscoveryClient
public class Application {
    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

（3）application.properties 增加路由的配置。

其中 order-service 替换成任何名称都可以，只要 path 和 serviceId 成对存在即可。这样所有/ product-service/**过来的请求都会转发到 product-service 上，所有 order-service/**过来的请求都被转发到 order-service。

还可以通过 url 来做映射，相对于 serviceId 的方式来说更复杂一些，因为需要知道具体的 IP 地址，不如直接引用实例名方便。

```
spring.application.name=zuul-service
server.port=8765
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/
# routes to serviceId
zuul.routes.product-service.path=/product-service/**
```

```
zuul.routes.product-service.serviceId=product-service
zuul.routes.order-service.path=/order-service/**
zuul.routes.order-service.serviceId=order-service
# routes to url
zuul.routes.product-service-url.path=/product-service-url/**
zuul.routes.product-service-url.url=http://localhost:2221/
zuul.routes.order-service-url.path=/order-service-url/**
zuul.routes.order-service-url.url=http://localhost:3331/
```

第四步：将 eureka-server、zuul-service、product-service、order-service 这四个工程启动，在监控界面可以看到启动成功，如图 7-10 所示。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ORDER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:order-service:3331
PRODUCT-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:product-service:2221
ZUUL-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:zuul-service:8765

图 7-10　仪表盘图

然后分别单独验证两个服务 product-service、order-service 是否可以正常使用，经验证都可以访问，如图 7-11(a)、图 7-11(b)所示。

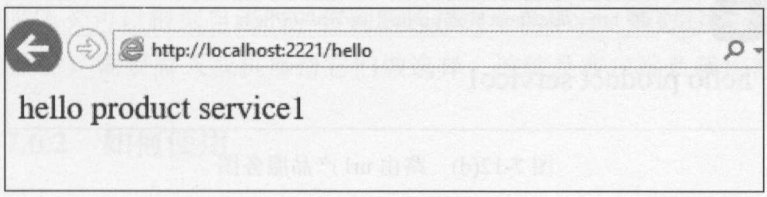


图 7-11(a)　产品服务图

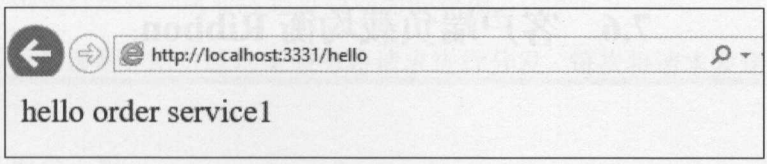


图 7-11(b)　订单服务图

在两个服务都验证成功之后就可以通过 Zuul 来验证路由的功能，可以看出

路由功能已经生效，可以正常地通过服务 ID 映射的方式进行跳转，如图 7-12(a)、图 7-12(b)所示。

同样可以通过 URL 映射的方式进行跳转，结果是一样的，如图 7-12(c)、图 7-12(d)所示。

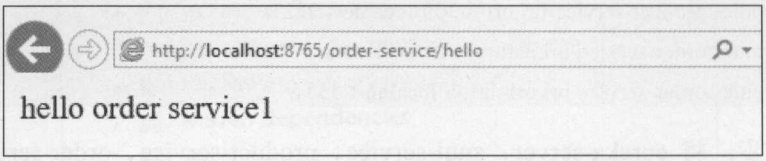


图 7-12(a)　路由订单服务图

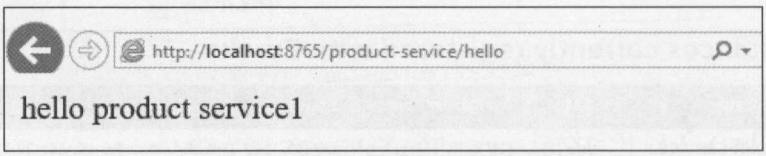


图 7-12(b)　路由产品服务图

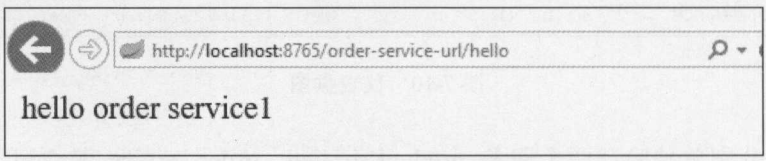


图 7-12(c)　路由 url 订单服务图

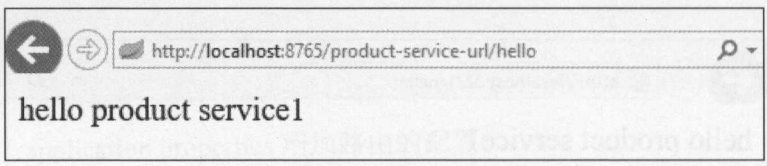


图 7-12(d)　路由 url 产品服务图

7.6　客户端负载均衡 Ribbon

Ribbon 是 Netflix 发布的开源项目，主要功能是提供客户端的软件负载均衡算法，将 Netflix 的中间层服务连接在一起。Ribbon 客户端组件提供一系列完善的配置项，如连接超时、重试等。

Ribbon 维护了一个服务列表，如果服务实例注销，Ribbon 能够自行将其剔除；但如果该服务实例重新启动，或者增加新的实例，需要手工调用 Ribbon 的接口把服务加进 Ribbon 的服务列表。这个工作还是交给 Eureka 好了，它可以做到服务列表的动态更新。

所以二者结合起来使用效果明显，Ribbon 提供了客户端负载均衡的功能，Ribbon 利用从 Eureka 中读取到的服务信息列表，在调用服务实例时，合理地进行负载。



7.6.1 负载均衡

客户端负载均衡，具体来说是通过采取某种策略为客户端选择一个服务端，从而尽可能减少对于服务端的压力，达到负载均衡的效果，是选择恐惧症患者的福音。

当请求面对同一服务的众多实例时就懵了，该选择哪个呢？

请求不会智能到自己知道选择哪个实例，它会始终如一地向同一实例不断地发起请求，这一个被请求的实例有分分钟要累死的节奏，其他实例无能为力，只有干瞪眼的份儿。

这时就需要有人替请求做出判断和选择，通过负载均衡的策略将请求分发给不同实例处理。

就像超市配有多个收银台，应用请求是排队结账的顾客，但是这些顾客是机器人，不像人类可以根据自己的观察判断选择人少的柜台，所以这些顾客每次都到同一个柜台，需要有人或机器帮它们做选择，这就是客户端负载均衡的作用。



7.6.2 如何使用

通过负载均衡服务来调用订单服务，验证负载均衡服务每次是否对订单服务的两个实例进行选择，从而达到负载均衡的作用。

请求处理流程：由负载均衡服务将请求进行分发，每次将请求发送到不同的实例上面。

总共四个工程：

eureka-server；

loadbalance-service；

order-service1；

order-service2。

第一步：创建 loadbalance-service 工程。

工程 loadbalance-service 目录结构，如图 7-13 所示。

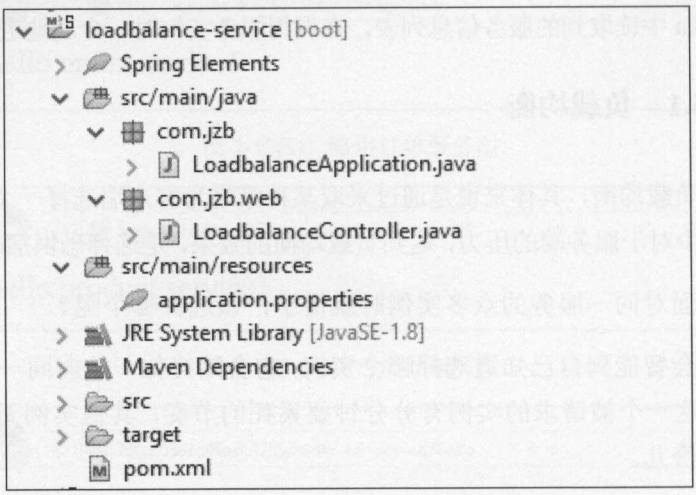


图 7-13　loadbalance-service 目录结构图

（1）启动代码。

加了注解@LoadBalanced 之后，这就变成了可以被注入到任意 service bean 的强化型 restTemplate 实例，通过一行注解就可以实现负载均衡的能力。@LoadBalanced 使用的是 spring-cloud-commons 包下的 org.springframework.cloud.lient.loadbalancer 的 LoadBalanced 接口。

代码如下。

```
@SpringBootApplication
@EnableDiscoveryClient
public class LoadbalanceApplication {
    @Bean
    @LoadBalanced
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
    public static void main(String[] args) {
```

```
        SpringApplication.run(LoadbalanceApplication.class, args);
    }
}
```

（2）在 Rest 方法上增加两个路径的映射，分别对应产品服务和订单服务，这样就可以根据不同的请求地址，将请求分发给不同的服务，继而通过负载均衡来调用不同的实例。

```
@RestController
public class LoadbalanceController {
    @Autowired
    RestTemplate restTemplate;
    @RequestMapping(value = "/product/hello", method = RequestMethod.GET)
    public String productHello() {
        return restTemplate.getForEntity("http://PRODUCT-SERVICE/hello", String.class).
getBody();
    }

    @RequestMapping(value = "/order/hello", method = RequestMethod.GET)
    public String orderHello() {
        return  restTemplate.getForEntity("http://ORDER-SERVICE/hello",  String.class).
getBody();
    }
}
```

第二步：复制两个订单工程 order-service1、order-service2 工程。

（1）order-service1 和 order-service2 端口设置成不一样的：

order-service1 端口 3331；

order-service2 端口 3332。

（2）显示结果不一样以便于验证每次请求的实例不同。

order-service1 显示 hello order service1。

```
    @RequestMapping(value = "/hello" ,method = RequestMethod.GET)
    public ResponseEntity<String> hello() {
      return new ResponseEntity<>("hello order service1", HttpStatus.OK);
    }
```

order-service2 显示 hello order service2。

```
@RequestMapping(value = "/hello" ,method = RequestMethod.GET)
 public ResponseEntity<String> hello() {
   return new ResponseEntity<>("hello order service2", HttpStatus.OK);
 }
```

第三步：将四个工程都启动之后，可以在监控页面看到状态，如图 7-14 所示。然后通过 loadbalance-service 的服务地址调用 Order 服务，查看每次调用的结果有何不同。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
LOADBALANCE-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:loadbalance-service:8763
ORDER-SERVICE	n/a (2)	(2)	UP (2) - 192.168.254.86:order-service:3332 , 192.168.254.86:order-service:3331

图 7-14　仪表盘图

第四步：通过访问http://localhost:8763/order/hello来验证，每次都会调用不同的实例，以此达到负载均衡的效果。

第一次结果，如图 7-15(a)所示。

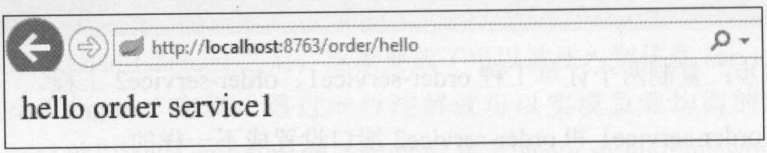


图 7-15(a)　负载均衡（第一次结果）

第二次结果，如图 7-15(b)所示。

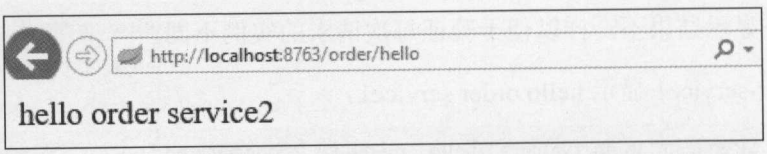


图 7-15(b)　负载均衡（第二次结果）

后台日志部分内容：

ORDER-SERVICE.ribbon.ActiveConnectionsLimit to use NEXT property:

niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647

Flipping property: ORDER-SERVICE.ribbon.ActiveConnectionsLimit to use NEXT property: niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647

c.n.l.DynamicServerListLoadBalancer : DynamicServerListLoadBalancer for client ORDER-SERVICE initialized: DynamicServerListLoadBalancer: {NFLoadBalancer:name=ORDER-SERVICE,current list of Servers= [192.168.254.117:3332, 192.168.254.117:3331],Load balancer stats=Zone stats: {defaultzone=[Zone:defaultzone; Instance count:2; Active connections count: 0; Circuit breaker tripped count: 0; Active connections per server: 0.0;]

},Server stats: [[Server:192.168.254.117:3332; Zone:defaultZone; Total Requests:0; Successive connection failure:0; Total blackout seconds:0; Last connection made:Thu Jan 01 08:00:00 CST 1970; First connection made: Thu Jan 01 08:00:00 CST 1970; Active Connections:0; total failure count in last (1000) msecs:0; average resp time:0.0; 90 percentile resp time:0.0; 95 percentile resp time:0.0; min resp time:0.0; max resp time:0.0; stddev resp time:0.0]

, [Server:192.168.254.117:3332; Zone:defaultZone; Total Requests:0; Successive connection failure:0; Total blackout seconds:0; Last connection made:Thu Jan 01 08:00:00 CST 1970; First connection made: Thu Jan 01 08:00:00 CST 1970; Active Connections:0; total failure count in last (1000) msecs:0; average resp time:0.0; 90 percentile resp time:0.0; 95 percentile resp time:0.0; min resp time:0.0; max resp time:0.0; stddev resp time:0.0]

]}ServerList:org.springframework.cloud.netflix.ribbon.eureka.DomainExtractingServerList@5c252085

通过日志可以看出当负载均衡发出请求，会根据服务名到注册中心上把所有实例都找到，保存到服务列表当中，这样即使注册中心宕机了，由于缓存了服务列表，也不影响正常使用。

当其中一个实例注销或重启之后，由于服务列表的更新需要时间，这个时间和服务本身设定的失效间隔有关，设置时间越短，从注册中心剔除的时间也越短，这样更新服务列表就会更加迅速，能保证请求不再去调用已经停

止的服务实例。

而实际中请求一般不会直接发送到负载均衡服务上面，在它前面需要有一个网关，这时我们在现在的工程基础之上集成网关，就相当于搭积木，在前面放了一个组件，不需要改变任何现有工程，只需要在网关里面加上路由映射即可。

下面验证网关 Zuul+负载均衡。

请求处理流程：增加网关服务，调用方不再是从负载均衡服务直接发起，改为由网关 zuul-service 路由到负载均衡服务上，然后负载均衡再将请求进行分发，发送到不同的实例上面。

第五步：增加 zuul-service 工程，在配置里增加对负载均衡的路由，然后启动工程，在监控界面看到已经成功启动，如图 7-16 所示。

```
zuul.routes.loadbalance-service.path=/loadbalance-service/**
zuul.routes.loadbalance-service.serviceId=loadbalance-service
```

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
LOADBALANCE-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:loadbalance-service:8763
ORDER-SERVICE	n/a (2)	(2)	UP (2) - 192.168.254.86:order-service:3332 , 192.168.254.86:order-service:3331
ZUUL-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:zuul-service:8765

图 7-16　仪表盘图

第六步：这时就可以通过 Zuul 来测试负载均衡的功能，通过访问 Zuul 的 8765 端口，并且加上 loadbalance-service 的路由地址，发现每次请求的都是不同的 order-service 的实例。

第一次请求结果，如图 7-17(a)所示。

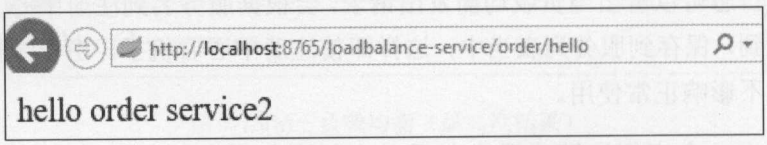


图 7-17(a)　zuul-lb-订单（第一次请求结果）

第二次请求结果，如图 7-17(b)所示。

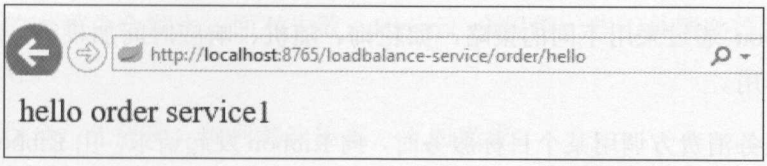


图 7-17(b)　zuul-lb-订单（第二次请求结果）

第七步：此时再增加 product-service1 的启动，看一下 Zuul 能否将请求通过负载均衡最终发送到 product-service 上，如图 7-18 所示。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
LOADBALANCE-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:loadbalance-service:8763
ORDER-SERVICE	n/a (2)	(2)	UP (2) - 192.168.254.86:order-service:3332 , 192.168.254.86:order-service:3331
PRODUCT-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:product-service:2221
ZUUL-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:zuul-service:8765

图 7-18　仪表盘图

第八步：通过/product/hello 来访问，经测试也可以正常调用 product-service，如图 7-19 所示。

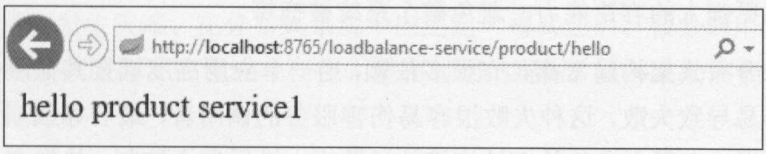


图 7-19　zuul-lb-产品图

至此客户端负载均衡的使用和验证也就结束了。



7.6.3　补充说明

好的选择比努力更重要。选对了，就是人生赢家，选错了，只剩背心裤衩。

如何选择是负载均衡的核心，通过最优的选择可以大幅减少系统的压力，反之则会让系统随时有崩溃的风险。

Ribbon 通过采用不同的策略，如轮询、随机、响应时间加权等，起到了调度员的作用。

当服务消费方调用某个目标服务时，向 Ribbon 发起请求，由 Ribbon 以某种策略做负载均衡后将请求转发到目标服务。

在多个实例的情况下，当一个实例“挂掉”，Ribbon 不会再访问这个实例。

当“挂掉”的实例又被重启了，Ribbon 会再找到这个实例，但是有时间间隔，大概是发送心跳的时间间隔。

当注册中心“挂掉”之后，Ribbon 不受影响，它可以继续进行正常的请求服务。因为缓存了服务列表，可以暂时使用。

这时如果其中一个实例“挂掉”了，那么 Ribbon 还会请求这个实例，因为无法获取更新的列表，所有注册中心可以暂时“挂掉”，但是不能长时间宕机。

7.7 断路器 Hystrix

安全第一，安全第一，安全第一，重要的事情说三遍。

熔断器 Hystrix 是容错管理工具，作用是通过隔离、控制服务从而对延迟和故障提供更强大的容错能力，避免整个系统被拖垮。

复杂分布式架构通常都具有很多依赖，当一个应用高度耦合其他服务时非常危险且容易导致失败，这种失败很容易伤害服务的调用者，最后导致一个接一个的连续错误，应用本身就处在被拖垮的风险中，最后失去控制，就像在一个高流量的网站中，某个单一的后端一旦发生延迟，将会在数秒内导致所有应用资源被耗尽。如何处理这些问题是有关系统性能和效率的关键性问题。

当在系统高峰时期，大量对微服务的调用可能会堵塞远程服务器的线程池，如果这个线程池没有和主应用服务器的线程池隔离，就可能导致整个服务器挂机。

Hystrix 使用自己的线程池，这样和主应用服务器线程池隔离，如果调用花费很长时间，会停止调用，不同的命令或命令组能够被配置使用它们各自的线程池，可以隔离不同的服务。

失败的时候总会发生，面对失败，Hystrix 会提供一些工具帮助技术人员更从容地应对。

要想保全，先隔离。如何解决隔离，下面来说。



7.7.1 如何隔离

（1）Hystrix 使用命令模式 HystrixCommand 包装依赖调用逻辑，每个命令在单独线程中执行。

（2）可配置超时的阈值，超过阈值时，直接返回或执行 fallback 逻辑。

（3）为每个依赖提供一个小的线程池，如果线程池已满则调用将被立即拒绝，默认不采用排队，加速失败判定时间。

（4）调用结果失败，包括异常、拒绝、超时、短路时执行 fallback（降级）逻辑。

（5）提供熔断器组件，可以自动运行或手动调用。

（6）提供实时的统计和监控。



7.7.2 电路熔断器模式

熔断机制的原理很简单，像家里的电路熔断器，如果电路发生短路能立刻熔断电路，避免发生灾难。在分布式系统中应用这一模式之后，服务调用方可以自己进行判断某些服务反应慢或者存在大量超时的情况时，能够主动熔断，防止整个系统被拖垮。

不同于电路熔断只能断不能自动重连，Hystrix 可以实现弹性容错，当情况好转之后，可以自动重连。这就好比魔术师把鸽子变没了容易，但是真正考验技术的是如何把消失的鸽子再变回来。

通过断路的方式，可以将后续请求直接拒绝掉，一段时间之后允许部分请求通过，如果调用成功则回到电路闭合状态，否则继续断开。

熔断器可以实现快速失败，如果它在一段时间内侦测到许多类似的错误，会强迫其以后的多个调用快速失败，也就是不让它们再次访问远程服务器了，这是通过 HystrixCommand 的 fallbackMethod 实现的，在过一段配置的时间后，

它会再次关闭熔断，又开始接受远程调用，然后再次侦测错误，再次熔断，如此反复。

7.7.3 回滚降级

正常情况下外部发送请求调用服务的 API，如图 7-20 所示。

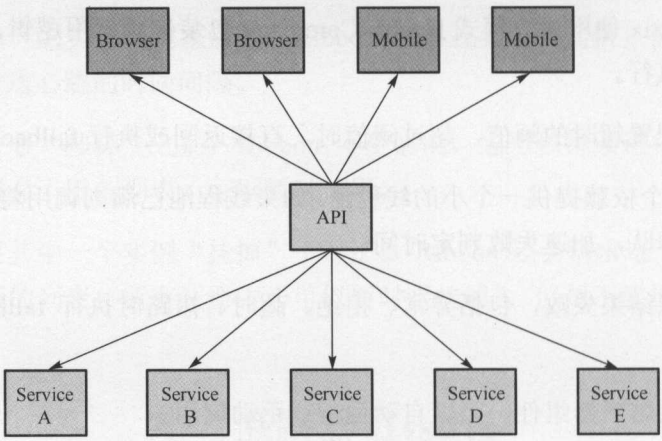


图 7-20 服务调用图

当有服务不能正常工作时，则直接调用 Fallback 方法，如图 7-21 所示。

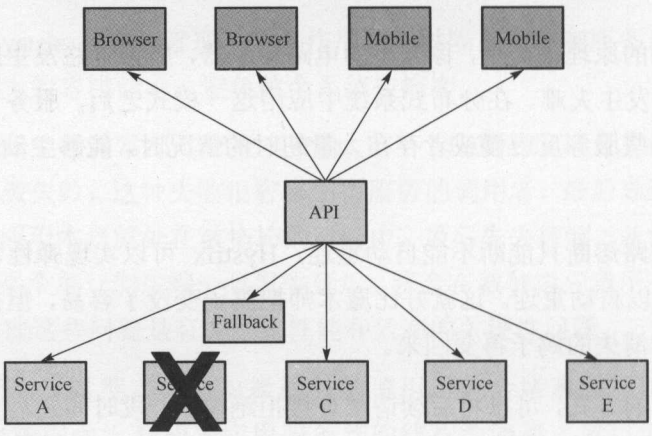


图 7-21 服务调用回滚图

当系统上线时，如果遇到问题导致无法使用，那么最简单直接的处理方式就是版本回退。

在熔断或限流发生时，也采用 fallback 的方式对请求进行快速处理，是一种比较可靠的手段。在回滚方法中可以返回一些缺省值或备份数据，让客户感知方面更友好一些。

虽然服务不可用，但是系统通过调用备用方法，给出一个友好的提示，虽然拒绝了用户的访问，但是会告诉用户一个结果。

平时去超市买活鱼，超市会提供杀鱼服务。但是到了节假日，超市买鱼的人特别多，工作人员忙不过来，资源紧张，都开始排队了，这个时候就不提供杀鱼服务了，这就是服务降级。因为主要服务是提供鱼，像杀鱼这种服务就是附带的服务。

系统也是这样，再比如京东 618、天猫双 11 时，系统可以做适当的服务降级，更好地保证系统的正常进行。

在微服务架构中如果遇到服务资源紧张等情况，也可以通过相关组件进行服务的降级操作，这样可以尽可能地保证主要业务的正常使用，而让边缘服务进行适当的让路。或者优先保证重点业务主体的使用，对一些不太重要的业务主体可以有选择性地放弃。

说完降级再顺便介绍一下服务升级，在软件系统里目前还没有服务升级这一概念，只有服务降级。服务升级就如同酒店常客计划里能得到行政套间的升级、移动流量套餐加大等。

在系统中如果增加一些服务升级带来的客户体验会更好，如建立一些 VIP 通道，可以快速审批，或者是做一些备选方案，当资源紧张时优先满足大客户的需要。



7.7.4 如何使用

通过 Netflix Hystrix 实现断路器容错的功能，使用 Fallback 方法为熔断或异常提供备用方案进行处理。涉及三个工程：eureka-server、product-service、order-service。

验证使用 order-service 调用 product-service，正常情况下返回产品服务的结果，当产品服务无法访问时，立刻调用回滚方法返回结果。

order-service 工程目录结构，如图 7-22 所示。

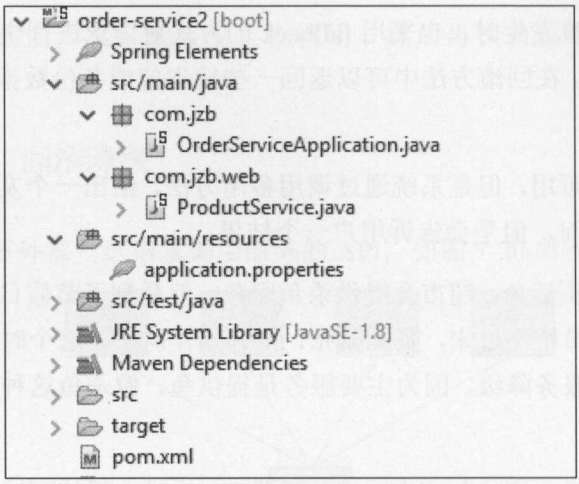


图 7-22　order-service 目录结构图

（1）pom.xml 增加 Hystrix 依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>
```

（2）启动应用增加注解@EnableCircuitBreaker，开启断路器功能。

```
@EnableDiscoveryClient
@SpringBootApplication
@RestController
@EnableCircuitBreaker
public class OrderServiceApplication {
    @Bean
    @LoadBalanced
    RestTemplate restTemplate() {
        return new RestTemplate();
    }

    public static void main(String[] args) {
        new SpringApplicationBuilder(OrderServiceApplication.class).web(true).run(args);
    }
```

```
    @RequestMapping(value = "/hello" ,method = RequestMethod.GET)
    public ResponseEntity<String> hello() {
        return new ResponseEntity<>("hello order service2", HttpStatus.OK);
    }

    @Autowired
    private ProductService productService;
    @RequestMapping(value = "/product/hello", method = RequestMethod.GET)
    public String helloProductService() {
        return productService.helloService();
    }
}
```

（3）增加 ProductService 类，通过它来调用 product-service，当调不通时立刻调用回滚方法。

```
@Service
public class ProductService {
    @Autowired
    RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "productServiceFallback")
    public String helloService() {
        return restTemplate.getForEntity("http://PRODUCT-SERVICE/hello", String.class).
getBody();
    }
    public String productServiceFallback() {
        return "product 已沽清";
    }
}
```

三个工程都启动之后可以在监控页面看到，如图 7-23 所示。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONFIG-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:config-service:5551
ORDER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:order-service:3332
PRODUCT-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:product-service:2221

图 7-23　仪表盘图

这时通过 order-service 可以正常调用 product-service，并且返回结果，如图 7-24 所示。

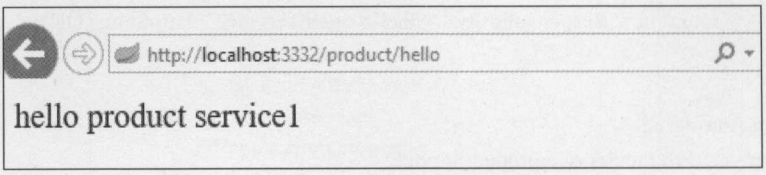


图 7-24　断路器正常使用图

现在停掉 product-service，看监控页面已经没有 product-service 了，如图 7-25 所示。

这时再次请求访问会得到回滚方法的结果，如图 7-26 所示。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONFIG-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:config-service:5551
ORDER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:order-service:3332

图 7-25　仪表盘图

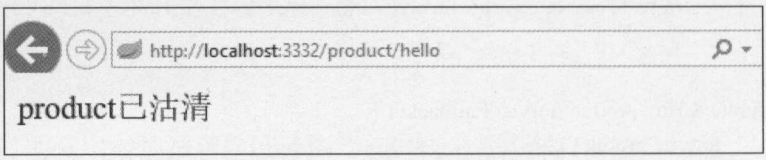


图 7-26　断路器异常回滚图

下面介绍使用 FeignClient 实现断路器的功能。一共三个工程：eureka-server、order-service、product-service。

和上面使用有一些不同，因为 FeignClient 里面已经包含了 Hystrix，所以不需要增加 Hystrix 的依赖，也不需要开启@EnableCircuitBreaker。

order-service 的目标结构，如图 7-27 所示。

（1）启动类代码如下。

```
@EnableDiscoveryClient
@SpringBootApplication
```

```
@RestController
@EnableFeignClients
public class OrderServiceApplication {
    @Autowired
    ProductServiceFeignDAO   productServiceFeignDAO;

    public static void main(String[] args) {
        new SpringApplicationBuilder(OrderServiceApplication.class).web(true).run(args);
    }

    @RequestMapping("/")
    public String home() {
        return "Hello Order Service World";
    }

    @RequestMapping(value = "/hello" ,method = RequestMethod.GET)
    public ResponseEntity<String> hello() {
        return new ResponseEntity<>("hello order service1", HttpStatus.OK);
    }

    @RequestMapping(value = "/product/hello", method = RequestMethod.GET)
    public String getHelloFromProductService() {
        return productServiceFeignDAO.hello();
    }
}
```

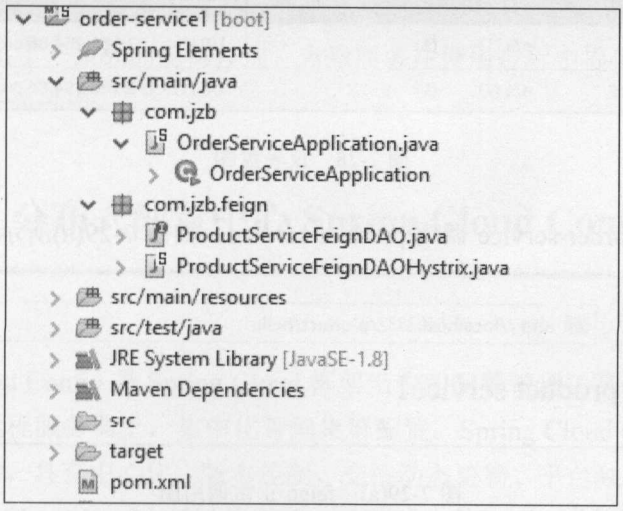


图 7-27　order-service 目录结构图

（2）增加接口 ProductServiceFeignDAO，用于调用 product-service，里面增加 fallback 配置，里面配置回滚接口。

```
@FeignClient(value = "product-service", fallback = ProductServiceFeignDAOHystrix.class)
public interface ProductServiceFeignDAO {

    @RequestMapping(method = RequestMethod.GET, value = "/hello")
    String hello();
}
```

（3）增加回滚接口，需要实现 ProductServiceFeignDAO 接口。

```
@Componentp
public class ProductServiceFeignDAOHystrix implements ProductServiceFeignDAO {
    @Override
    public String hello() {

        return "服务不见鸟，稍后再试";
    }
}
```

将三个工程启动之后，在监控页面看启动状态，如图 7-28 所示。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ORDER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:order-service:3332
PRODUCT-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:product-service:2221

图 7-28　仪表盘图

然后通过 order-service 调用 product-service，如图 7-29(a)所示。

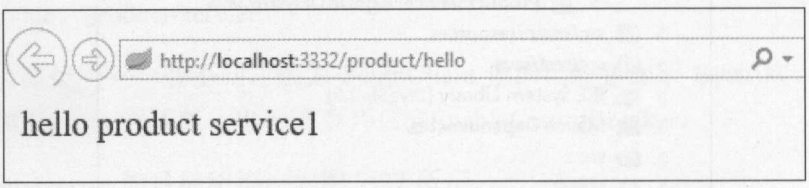


图 7-29(a)　feign 正常调用图

这时停掉 product-service，再调用 product-service 时验证是否调用断路器回滚方法，发现已经触发了回滚的方法，如图 7-29(b)所示。

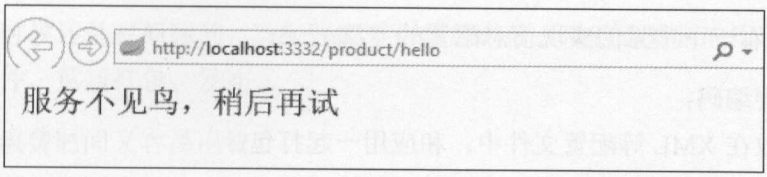


图 7-29(b)　feign 异常调用图

7.7.5　交通疏导

在大城市遭遇堵车是家常便饭的事，有时堵了好久，最终发现只是一起小的剐蹭事故引起的。

而且这种事故会导致不仅这一条路发生拥堵，连其他的路也会受影响，故障漫延非常厉害。

传统应用也有类似情况，一个小故障会在极短时间拖垮整个系统。

所以解决拥堵，不仅需要快速处理事故，而且还要把其他路上正在向这条路上汇聚的车及时导流到其他道路，不能让车再往这条路上行驶，相当于系统里直接拒绝当前应用的请求。

7.7.6　小结

Hystrix 提供了一系列的防御技能，为微服务在使用过程中提供了安全保障，从而让系统实现高可用。

7.8　分布式配置中心 Spring Cloud Config

Spring Cloud Config 是 Spring Cloud 框架当中的配置管理工具包，让用户可以把配置放到远程服务器上，集中化管理集群配置。Spring Cloud Config 是云端存储配置信息的，具有中心化、版本控制、支持动态更新、平台独立、语言独立等特性。Spring Cloud Config 可以从 SVN、GIT 或文件资源中获取配置，并支持动态更新，无须重启等特性。

下面看一下常见的实现信息配置的方法：

- 硬编码；
- 放在 XML 等配置文件中，和应用一起打包；
- 文件系统中；
- 云端存储。

然而硬编码的缺点是代码与配置耦合，需要修改代码，工作量大、风险大。放在 xml 或 properties 等配置文件中的缺点是修改之后需要重新打包和重启。保存在文件系统中的缺点是依赖操作系统。云端存储需要单独增加服务。



7.8.1 为什么要有配置

一个应用程序一般是一个应用包，如 ear 包、war 包、jar 包等。

这个包里 99%以上是代码，剩余部分一般是配置文件，也就是说程序是由代码+配置构成的。

应用中的这些配置文件，又分必要的和非必要的。

必要的配置文件中存放了一些基本配置，如数据库配置、国际化、定时任务、日志配置等，还会有一些地址的配置如 server 和端口、应用名称。应用中会经常见到的 xml、properties、yml 等就是用来存储这些配置信息的。

非必要的配置文件如 spring 的 XML 配置已经逐渐被 annotation 所取代。

为什么要把这些配置放到配置文件中呢？因为这些配置是可能变化的，不能在程序中硬编码。以后一旦改起来很麻烦，所要放在一起统一管理。

这时我们会进行第一次提取配置的操作，将配置信息统一放到配置文件中。

- 一次提取配置

代码和配置是要分开的，因为二者的属性是不同的，不应该耦合在一起。

将这些非代码的配置，抽取到配置文件中，统一管理。

这是第一次抽取的目的，初步效果是解除了配置和代码之间的耦合关系，更方便管理配置。

这时候还是有问题的，当配置需要更新的时候，我们需要到应用程序里修改配置文件，重新打包、发布。

既浪费时间又容易出错。

➢ 二次提取配置

使用传统应用时一般通过一次抽取就可以了，但是在微服务架构下，有成百上千个应用的时候，不能再像以前那样，到每个应用下面去修改文件了，这样修改的地方太多，容易出错，这时候就需要专门的配置中心服务来统一管理配置，这就是二次抽取。

将所有应用的配置抽取出来放在一个服务器上，所有的服务通过统一的配置中心服务来读取各自的配置，这样就可以在一个地方统一维护这些配置。

当有配置更新的时候，只需要执行相关命令就可以将应用的配置进行同步，这就省去了打包、重启的过程。



7.8.2 集中配置的好处

传统应用上的发包步骤是解压应用包，把配置文件拽出来，修改完再压回去，再重新部署，很容易出错。

如果每个服务都直接从服务器上读取配置，那么一旦服务器做迁移，所有的服务都要去修改服务器的地址，如果服务很多又是一场灾难。

所以通过配置中心做一层隔离，就可以避免这种情况的发生，如果一旦发生服务器变更等情况，无论是地址变了，还是服务器类型从 GIT 变成 SVN 都只需要修改配置中心一处的配置就可以了。不影响服务的正常使用，保证了服务的高可用性。



7.8.3 配置方式的演进

早期因为开发环境的数据库和测试环境、生产环境是不一样的，在交付测试和生产的时候都要通过手工的方式更改配置，作者曾经遇到过一个项目每次上线需要改 5 个配置文件，当要改动的地方越多时越容易出错，这也是为什么要采用集中配置，在一个地方统一修改的原因。而且要尽量减少这种人为干预的方式，因为容易出错。

后来 spring 应用利用策略方法的设计模式，将不同环境对应不同的配置文件

统一打到一个应用包里，分别对应-dev、-local、-uat、prd 等，这样在不同环境使用时只需要修改主配置文件当中的 profile 参数来指定当前环境即可，相当于通过开关来设置生存模式，只修改一个地方，就可以了。

现在的情况更复杂了，微服务架构下原来的一个应用变成了 *N* 个应用，对应的配置方式又要进行升级换代，从而引出了集中配置 Spring Cloud Config。



7.8.4 如何使用

Spring Cloud Config 目前支持三种存储方式：本地资源、SVN 和 GIT。原理介绍如下。

首先在 GIT 或本地环境上增加配置文件，配置文件的命名规则为“应用名+环境名+格式”，如下面的例子。

eg: orderservice-dev.properties, orderservice-local.properties

在 server 端配置 GIT 或本地环境的地址，server 可以访问 GIT 或本地环境上的资源。由 server 集中管理配置。

client 端配置 server 的地址，通过调用 server 获取 GIT 或本地环境上的资源。多个 client 调用同一个 server，而不是每个 client 直接访问服务器上的资源，相当于通过 server 做了一层隔离，通过 server 做代理，避免资源文件地址变更或者换成其他存储方式实现时，所有的 client 不得不同时更改配置项。

同时 config 还支持集群式的部署方式，利于扩展。如图 7-30 所示。

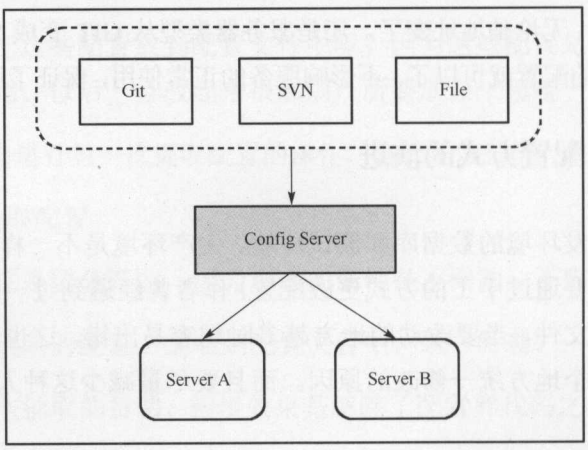


图 7-30 配置服务图

先通过本地资源存储的方式示范如何使用。

第一步：创建配置服务，其他服务通过配置服务读取本地配置文件的内容。

（1）首先创建工程配置服务中心 config-service，目录结构如图 7-31 所示。

（2）服务端 pom.xml 增加如下内容。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

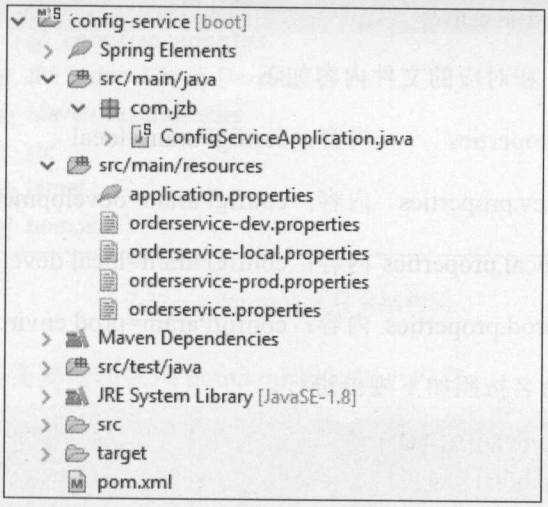


图 7-31　config-service 目录结构图

（3）启动类增加@EnableConfigServer 注解，开启 Config Server 的功能。代码如下。

```
@EnableConfigServer
@SpringBootApplication
@EnableDiscoveryClient
public class ConfigServiceApplication {
    public static void main(String[] args) {
        new SpringApplicationBuilder(ConfigServiceApplication.class).web(true).run(args);
    }
}
```

（4）application.properties 中配置注册地址及服务信息。

采用本地存储配置的方式通过设置属性 spring.profiles.active=native 来实现。

config-service 会默认从应用的 src/main/resource 路径下查找对应环境的配置文件。

```
spring.application.name=config-service
server.port=8888
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/
eureka.instance.lease-renewal-interval-in-seconds=50
eureka.instance.lease-expiration-duration-in-seconds=30
spring.profiles.active=native
```

（5）资源文件和对应的文件内容如下。

orderservice.properties　　内容：configParam=local

orderservice-dev.properties　内容：configParam=development

orderservice-local.properties 内容：configParam=local development

orderservice-prod.properties 内容：configParam=prod environment be careful

资源文件的命名按照如下规范进行。

```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

label 是可选的，如 trunk、branch 等。

（6）启动成功之后按下面的格式发起请求，结果如下。

http://localhost:8888/{appname}/{env}/{label}，其中 label 可以省略。

本例中为：http://localhost:8888/orderservice/prod/。

```
{"name":"orderservice","profiles":["prod"],"label":null,"version":null,"propertySources":[{"name":"classpath:/orderservice-prod.properties","source":{"envParam":"prod environment be careful"}},{"name":"classpath:/orderservice.properties","source":{"envParam":"local"}}]}
```

第二步：通过配置中心获取配置参数，也就是如何获取 configParam 的值。

（1）首先还是用订单服务来举例，让订单服务可以读取配置。目录结构如图 7-32 所示。

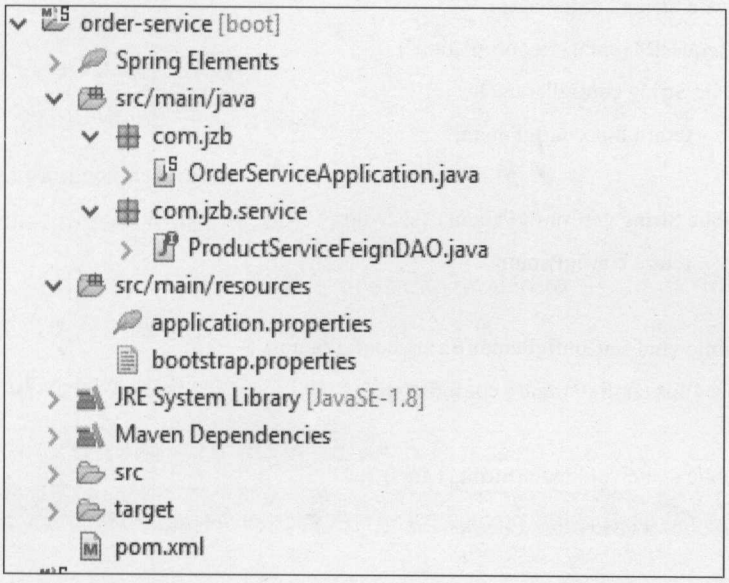


图 7-32　order-service 目录结构图

（2）订单服务需要增加依赖 pom.xml 增加如下内容。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

（3）启动应用代码如下：新增加了@RefreshScope 标签，和@Value("${configParam}")直接引用外部的配置或者通过 spring.profiles.active = ${config.name} 的方式引用。这样就可以通过 rest 请求的方式来验证参数是否获取成功。

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
@RefreshScope
@RestController
public class OrderServiceApplication {
    @Autowired
    ProductServiceFeignDAO productServiceFeignDAO;
```

```
    @Value("${configParam}")
    private String configParam;
    @RequestMapping("/configParam")
    public String configParam() {
        return this.configParam;
    }
    public String getConfigParam() {
        return configParam;
    }
    public void setConfigParam(String configParam) {
        this.configParam = configParam;
    }
    public static void main(String[] args) {
        SpringApplication.run(OrderServiceApplication.class, args);
    }


    @RequestMapping("/")
    public String home() {
        return "Hello Order Service World";
    }

    @RequestMapping(value = "/hello" ,method = RequestMethod.GET)
    public ResponseEntity<String> hello() {
        return new ResponseEntity<>("hello order service1", HttpStatus.OK);
    }

    @RequestMapping(value = "/product/hello", method = RequestMethod.GET)
    public String getHelloFromProductService() {
        return productServiceFeignDAO.hello();
    }

}
```

（4）bootstrap.properties 配置如下。

```
spring.application.name=orderservice
server.port=3331
spring.cloud.config.profile=local
spring.cloud.config.uri=http://localhost:8888/
```

spring.cloud.config.profile 像开关一样选择是哪个环境的配置，spring.cloud.config.uri 指向了配置服务的地址。

注意：上面这些属性必须配置在 bootstrap.properties 中，配置中心才能被正常加载。因为所有的配置服务是最先被加载进来的。

（5）当两个服务都启动之后可以在监控界面里看到，如图 7-33 所示。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONFIG-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:config-service:5551
ORDER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:order-service:3331

图 7-33　仪表盘图

（6）这时访问订单服务的/configParam 就可以看到，订单服务可以正确获取配置参数，如图 7-34 所示。

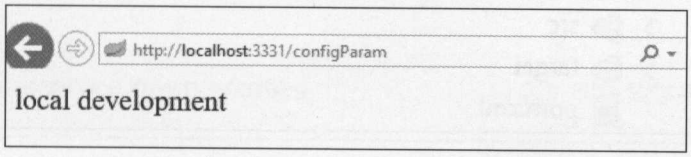


图 7-34　客户端使用配置服务图

上面讲了如何从本地获取配置，下面讲如何通过 GIT 获取配置。

第一步：创建 Git Server 端。

Config Server 端工程目录结构如图 7-35 所示。

（1）在 pom.xml 中增加依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
```

```
    </dependency>
    <dependency>
            <groupId>org.eclipse.jgit</groupId>
            <artifactId>org.eclipse.jgit</artifactId>
            <version>4.6.0.201612231935-r</version>
    </dependency>
    <dependency>
            <groupId>org.eclipse.jgit</groupId>
            <artifactId>org.eclipse.jgit.http.server</artifactId>
            <version>4.6.0.201612231935-r</version>
    </dependency>
```

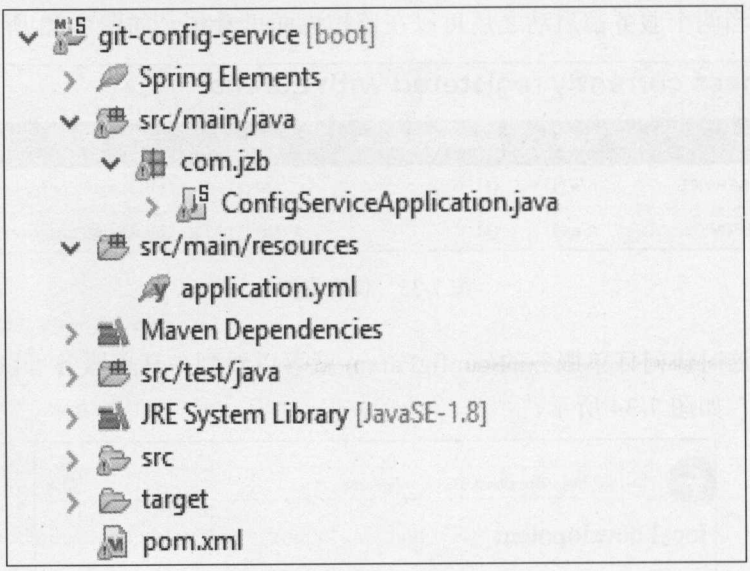


图 7-35　GIT 配置中心图

（2）在启动类代码中增加注解@EnableConfigServer 开启集中配置功能。

```
@EnableConfigServer
@SpringBootApplication
public class ConfigServiceApplication {
    public static void main(String[] args) {
        new SpringApplicationBuilder(ConfigServiceApplication.class).web(true).run(args);
    }
}
```

（3）配置文件 bootstrap.yml 做如下设置。

```
server:
  port: 8888
spring:
  cloud:
    config:
      server:
        git:
          uri: https://git.xxx.com/xxrepository/xxmachine.git
          searchPaths: config-repository
          username: xxx
          password: xxx
```

（4）在 GIT 上面创建 config-repository 目录，增加 orderservice-dev.properties 用于保存配置信息，如图 7-36 所示。

文件内容：configParam: dev env from git，如图 7-37 所示。

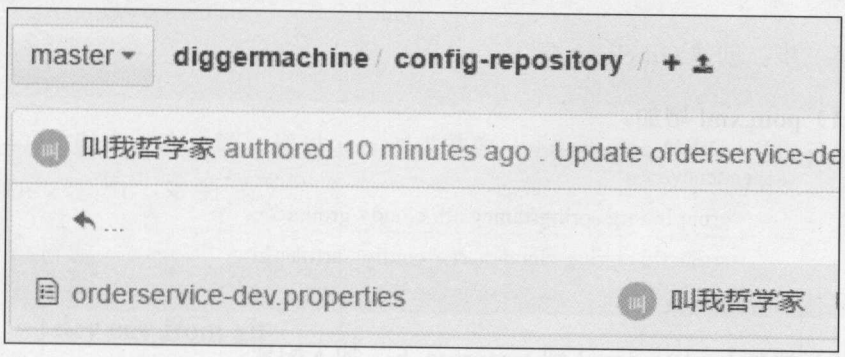


图 7-36　GIT 目录图

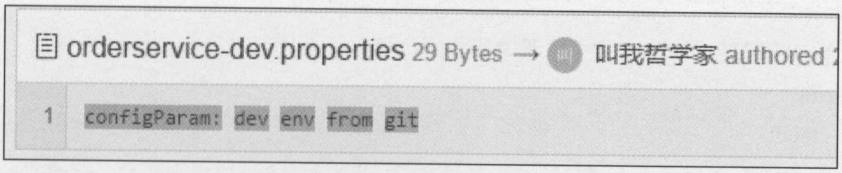


图 7-37　GIT 配置文件图

（5）这时可以通过如下地址可以访问资源文件。

http://localhost:8888/orderservice/dev

```
{
    "name": "orderservice",
    "profiles": [
        "dev"
    ],
    "label": null,
    "version": "aa67264040210bdffcec14ce3026eafb40871483",
    "propertySources": [
        {
            "name":    "https://git.xxx.com/xxrepository/xxmachine.git/config-repository/
orderservice-dev.properties",
            "source": {
                "configParam": "dev env from git"
            }
        }
    ]
}
```

第二步：创建 Git client 端。

（1）pom.xml 增加。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

（2）增加 bootstrap.yml 到 resources 下，加入配置。

```
server:
  port: 2221
spring:
  cloud:
    config:
      name: orderservice
      profile: dev
      label: trunk
      uri: http://localhost:8888
```

（3）代码如下。

```
@SpringBootApplication
@RefreshScope
@RestController
public class Application {
      public static void main(String[] args) {
            new SpringApplicationBuilder(Application.class).web(true).run(args);
      }
      @Value("${configParam}")
      private String configParam;
      @RequestMapping("/configParam")
      public String configParam() {
          return this.configParam;
      }
      public String getConfigParam() {
            return configParam;
      }
      public void setConfigParam(String configParam) {
            this.configParam = configParam;
      }
}
```

（4）这时启动客户端，通过 rest 请求获取 configParam 的值，发现已经可以正常获取，如图 7-38 所示。

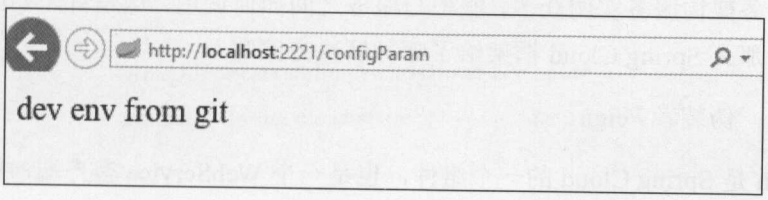


图 7-38　客户端获取配置图

（5）这时在 GIT 上对文件内容进行更新，变成如下内容。

```
configParam:dev env from git update
```

（6）此时访问http://localhost:8888/orderservice/dev 发现内容已经更新过来了。

```
{
    "name": "orderservice",
    "profiles": [
```

```
            "dev"
        ],
        "label": null,
        "version": "aa67264040210bdffcec14ce3026eafb40871483",
        "propertySources": [
            {
                "name": "https://git.xxx.com/xxrepository/xxmachine.git/config-repository/orderservice-
dev.properties",
                "source": {
                    "configParam": "dev env from git update"
                }
            }
        ]
    }
```

但是这时通过客户端调用结果仍然是之前的内容，没有更新。当配置更新之后可以通过 refresh 或 restart 方式发送请求进行更新，这样就不需要重启服务了。

7.9 服务之间调用 Feign

微服务架构服务实例众多，服务与服务之间如何调用，这是首先要面对的一个问题，那么 Spring Cloud 框架给了哪种解决方案呢？

答案：伪装者 Feign。

Feign 是 Spring Cloud 的一个组件，也是一个 WebService 客户端，用于服务之间的调用。



7.9.1 Feign 如何使用

通过伪装者 Feign 来实施服务之间的调用。

现在有一个服务叫“纽约”。

应用名： NEWYORK

第一步：创建一个普通 rest 服务。

```
@Service
@RestController
@RequestMapping(value = "/api/rest")
public class CommonWebService {
    @RequestMapping(value = "/baseInfo", method = RequestMethod.GET, produces =
MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<List<BaseInfoDTO>> getBaseInfo(){

        return new ResponseEntity<>("I am NEWYORK", HttpStatus.OK);
    }
```

第二步：application.yml 中配置它的应用名，也就是引用别名。

```
spring:
  application:
    name: NEWYORK-V1-1
```

至此被调用方的工程已经创建完毕，它没有任何特别之处，就是普通的服务。

下面我们要调用纽约的服务，创建一个服务，应用：假装在纽约

第一步：pom.xml 增加 Feign 的依赖。

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
```

第二步：增加一条@EnableFeignClients 注解，加上这个注解之后，相当于打开了伪装者客户端的开关，这个服务下面的接口就可以使用 feignclient 功能了。

```
@EnableFeignClients(basePackages = "com.xx.xxxservice")
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
```

第三步：创建一个接口用于调用真正的 NEWYORK。

接口上增加注解，注意这里的@FeignClient 要和被引用的 application 引用名一样。

```
@FeignClient("NEWYORK-V1-1")
public interface NewYorkFeignDAO {
    @RequestMapping(value = "/newyork/api/rest/baseInfo method = RequestMethod.GET,
produces = MediaType.APPLICATION_JSON_VALUE)
    List<xx> getXxxByFeignClient();
}
```

这样就可以实现服务之间的调用了。

再来看之前经常出现的产品服务和订单服务，订单服务会调用产品服务获取产品的信息，这两个服务之间存在调用关系，那么如何让订单服务调用产品服务呢？

首先被调用方不需要做任何修改，这个是 Spring Cloud 框架的一个很大的优点，被调用者只是一个工具，使用的时候直接拿来用，不需要双方都修改，只需要改一下调用方即可，相当于一个人要使用铁锹，这时只需要戴上手套即可，自身的改动也很小。

order-servcie 工程结构如图 7-39 所示。

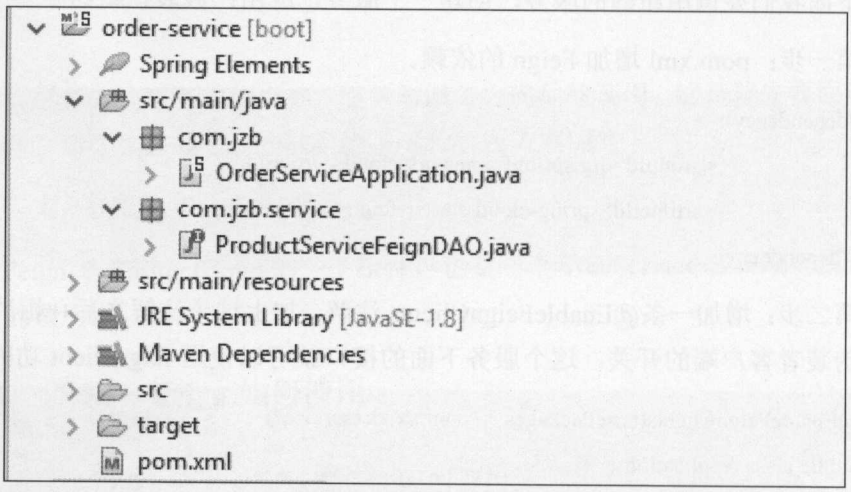


图 7-39 订单目录结构图

（1）pom.xml 增加 FeignClient 的依赖。

```
<dependency>
```

```
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
```

（2）增加接口，从命名上来看这个接口可以当成是工程本身的一个 DAO 来使用。第一行代码相当于引用了产品服务的别名。

```
@FeignClient("product-service")
public interface ProductServiceFeignDAO {

    @RequestMapping(method = RequestMethod.GET, value = "/hello")
    String hello();
}
```

（3）启动类增加@EnableFeignClients 开启 Feign 功能，新增 Rest 接口用于调用产品服务。

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
@RestController
public class OrderServiceApplication {
    @Autowired
    ProductServiceFeignDAO productServiceFeignDAO;
    public static void main(String[] args) {
        SpringApplication.run(OrderServiceApplication.class, args);
    }
    @RequestMapping("/")
    public String home() {
        return "Hello Order Service World";
    }
    @RequestMapping(value = "/hello" ,method = RequestMethod.GET)
    public ResponseEntity<String> hello() {
        return new ResponseEntity<>("hello order service1", HttpStatus.OK);
    }
     @RequestMapping(value = "/product/hello", method = RequestMethod.GET)
     public String getHelloFromProductService() {
```

```
        return  productServiceFeignDAO.hello();
    }
}
```

（4）这时当调用/hello 时调用的是 order-service 自己的服务，当路径是/product/hello 时调用的是 product-service，如图 7-40(a)、图 7-40(b)所示。

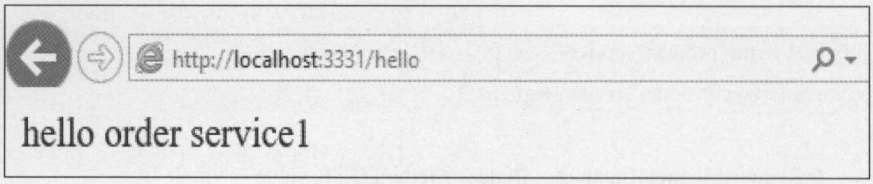


图 7-40(a)　订单服务

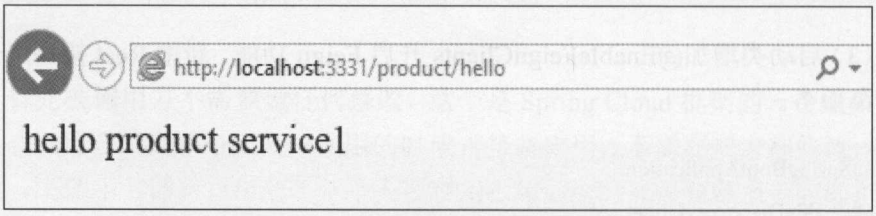


图 7-40(b)　订单服务调用产品服务

总的来说，服务之间的调用就是调用方首先引入 Feign 依赖，开启 Feign 功能，然后创建一个 Feign Client 接口用于调用其他服务。

传统的 Http Client 调用方式，需要进行构造请求、解析返回的数据等一连串的动作，十分麻烦。

Feign 很好地解决了服务之间远程调用的问题，直接通过接口方法进行调用，十分方便。

接口方法只是普通的 Rest 服务，不需要任何额外的配置。调用方只需要在启动类上增加一条注解，接口上引用被调用方的别名就可以，对于被调用方来说没有任何代码级的渗透，全程透明，和调用本地方法没有区别。

同时 Feign 内部集成了 Ribbon，所以它自带负载均衡功能。



7.9.2　补充说明

在多个实例的情况下，当一个实例“挂掉”，Feign 不会再访问这个实例，

不会报错。当“挂掉”的实例又被重启了，Feign 会再找到这个实例，时间上比 Ribbon 要快。

当注册中心“挂掉”之后，Feign 不受影响，它可以继续正常的请求服务。因为缓存了服务列表，可以暂时使用。

这时如果其中一个实例“挂掉”了，那么 Feign 还会请求这个实例，因为无法获取更新的列表，所有注册中心可以暂时“挂掉”，但是不能长时间宕机。

7.10 服务追踪

在微服务架构当中，要完成一个功能，通过 Rest 请求服务 API 调用服务来完成，整个调用的过程可能会聚合多个后台服务协同完成，在整个链路之上，任何一处调用超时或出错都可能造成前端请求的失败。

这时跟踪记录这些请求的调用的情况比以往要复杂得多，这就需要一个专门的工具来处理，Spring Cloud Sleuth 组件就是用于跟踪记录的工具。

Spring Cloud Sleuth 组件就像一个私家侦探，跟踪指定目标的所有请求，记录这些请求的所有操作，它到过哪些地方、驻留多久、接触过哪些人，并将这个过程用一条线索串起来，写到小本子上。

在系统里通过唯一 ID 保证服务调用时的一致性，记录请求从一个服务调用了哪些其他服务，能清楚地知道调用关系及每个调用环节的时长，出现问题时便于定位。

最后通过 Zipkin 进行日志聚合的操作，将所有记录信息收集整理，统一展示。



7.10.1 Spring Cloud Sleuth 介绍

Spring Cloud Sleuth 为 Spring Cloud 应用实现了一种分布式追踪解决方案，其兼容了 Zipkin 等工具。

Sleuth 的概念图如图 7-41 所示。

Sleuth 就相当于为微服务架构引入了一套记录体系，包含两部分：一个是 trace ID；另一个是 span ID。随时记录一个请求的每一步操作。

这个过程就像快递订单跟踪系统，一个快递单就是一个 trace ID，每到一个站点就对应一个唯一的 span ID，整个订单处理过程中的 trace ID 始终不变。

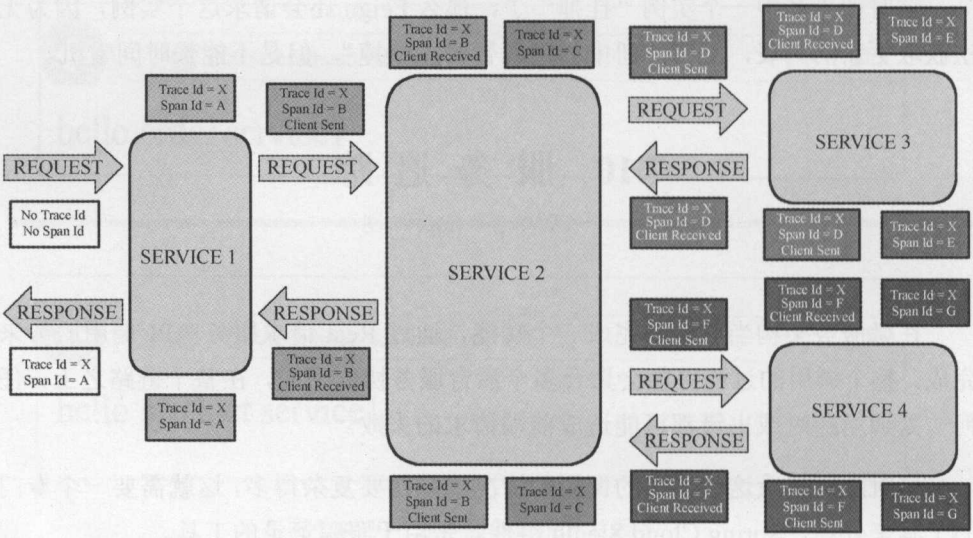


图 7-41　sleuth 示意图



7.10.2　Sleuth 如何使用

如果单独使用 Sleuth，那么只需要几个工程之间存在调用关系就可以。下面三个工程分别是负载均衡服务、订单服务、产品服务，它们之间的调用关系是负载均衡服务调用订单服务，订单服务调用产品服务。

使用 Sleuth 非常简单，只需要在 pom.xml 增加 Sleuth 的依赖，然后在调用方法上加 log 即可，系统会自动将跟踪信息加到 log 里面，打印出来。

三个工程如下。

loadbalance-service；

order-service；

product-service。

总共分三步。

第一步：首先在三个工程的 pom.xml 上增加如下依赖。

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

第二步：非常简单，每个工程调用方法上增加一条 log。

loadbalance-service 工程只需要在调用 order-service 的方法上增加下面一条 log。

```
@RequestMapping(value = "/order/hello", method = RequestMethod.GET)
public String orderHello() {
    log.info("loadbalance calling Order service");
    return restTemplate.getForEntity("http://ORDER-SERVICE/hello", String.class).getBody();
}
```

order-service 工程只需要在调用 product-service 的方法上增加下面一条 log。

```
@RequestMapping(value = "/hello", method = RequestMethod.GET)
public String orderHello() {
    log.info("order service calling product service");
    return restTemplate.getForEntity("http://PRODUCT-SERVICE/product/hello",
        String.class). getBody();
}
```

product-service 工程只需要在方法上增加下面一条 log。

```
@RequestMapping(value = "/product/hello" ,method = RequestMethod.GET)
public ResponseEntity<String> hello() {
    log.info("called by order service");
    return new ResponseEntity<>("called by order service", HttpStatus.OK);
}
```

第三步：启动三个工程，全部启动成功之后，通过监控页面可以看到状态，如图 7-42 所示。

接下来通过 loadbalance-service 发起请求，查看结果，如图 7-43 所示。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
LOADBALANCE-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:loadbalance-service:8763
ORDER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:order-service:3332
PRODUCT-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:product-service:2221

图 7-42　仪表盘图

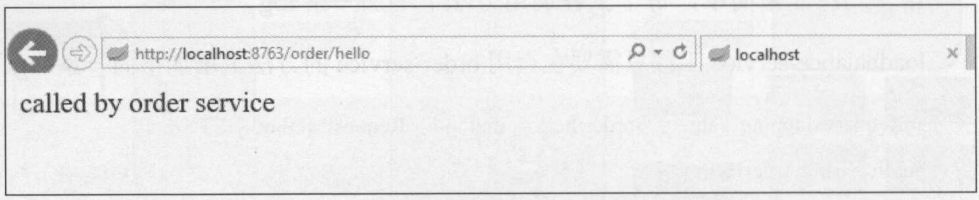


图 7-43　负载均衡请求图

可以看到调用结果返回正确，接下来看日志信息。这时看后台日志记录有三条信息，括号中分别是服务名、traceID 和 spanID。

可以看出 traceID 是同一个，而 spanID 每次都不同。这样通过 traceID 就可以查找出这一请求在执行过程中调用的所有环节，进而分析请求效率，定位问题等。

```
    INFO [loadbalance-service,6f9d2a44bd1dfd80,6f9d2a44bd1dfd80,false] 12392 ---
[nio-8763-exec-5] com.jzb.web.LoadbalanceController : loadbalance calling Order service
    INFO [order-service,6f9d2a44bd1dfd80,2ba79f3ea3c95ff2,false] 12476 --- [nio-3332-exec-3]
com.jzb.OrderServiceApplication : order service calling product service
    INFO [product-service,6f9d2a44bd1dfd80,53bd350025e65ec8,false] 11420 ---
[nio-2221-exec-3] com.jzb.ProductServiceApplication : called by order service
```



7.10.3　日志聚合 Zipkin

为了解决微服务全链路调用如何追踪、定位的问题，Google 发表了 Dapper 论文，介绍了他们如何进行服务追踪分析。他们的思路是在服务调用的请求和响应中加入唯一 ID，通过 ID 就可以明确上下游请求之间的关系。通过这些信息，结合可视化工具就可以很直观地分析出服务调用路径和服务间的依赖关系。

Zipkin 是 Dpper 的开源实现，支持多种语言，包括 JavaScript、Python、Java、Scala、Ruby、C#、Go 等。

上一步中 Spring Cloud Sleuth 已经将每个请求从开始调用到完成的每一步都进行了记录，但是这些 log 信息会很分散，使用起来不太方便，就需要有一个工具可以将这些信息进行收集和汇总，并且显示可视化的结果，便于分析和定位。

这时需要创建一个 Zipkin Server 用于收集和展示这些调用链路的信息，它的使用也很方便。

第一步：创建一个 Zipkin Server，正常启动，可以看到界面效果。

zipkin-service 工程目录如图 7-44 所示。

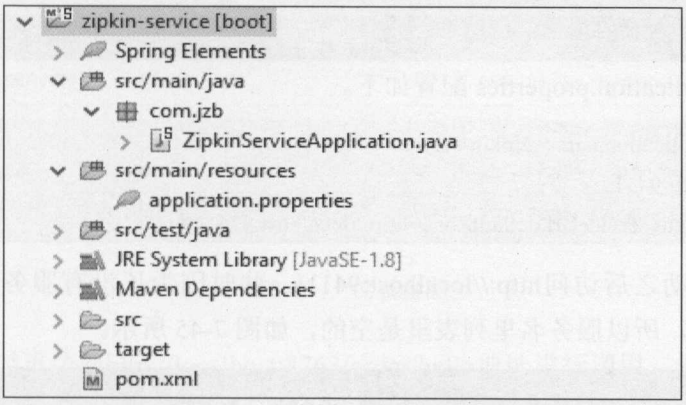


图 7-44　zipkin 目录结构图

（1）首先在 pom.xml 上增加依赖。

```
<dependency>
    <groupId>io.zipkin</groupId>
    <artifactId>zipkin-ui</artifactId>
    <version>1.39.3</version>
</dependency>
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-server</artifactId>
</dependency>
```

（2）ZipkinServiceApplication 增加@EnableZipkinServer，用来开启 Zipkin-Server 的功能。

代码如下。

```
@EnableDiscoveryClient
@EnableZipkinServer
@SpringBootApplication
public class ZipkinServiceApplication {
    private static Logger log = LoggerFactory.getLogger(ZipkinServiceApplication.class);

    public static void main(String[] args) {
        new
SpringApplicationBuilder(ZipkinServiceApplication.class).web(true).run(args);
    }
}
```

（3）application.properties 配置如下。

```
spring.application.name=zipkin-service
server.port=9411
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/
```

（4）启动之后访问http://localhost:9411/，此时因为还没有服务请求关联到zipkin server，所以服务名里列表里是空的，如图 7-45 所示。

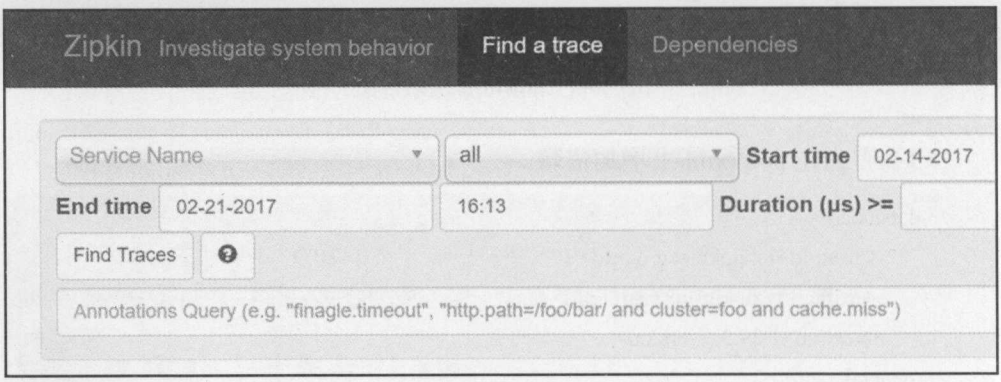


图 7-45　zipkin 界面图

第二步：想要在界面上能看到上一环节收集的日志信息及依赖关系，需要对上一步的每个工程中增加 sleuth 与 Zipkin 的依赖，然后增加注册地址，指向到zipkin server 上就可以了。

（1）首先每个工程的 pom.xml 增加一个依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

（2）接下来在配置文件 application.properties 中增加注册地址。

```
spring.zipkin.base-url=http://IP:9411/
```

（3）这时将四个工程全部启动，启动结果如图 7-46 所示。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
LOADBALANCE-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:loadbalance-service:8763
ORDER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:order-service:3332
PRODUCT-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:product-service:2221
ZIPKIN-SERVICE	n/a (1)	(1)	UP (1) - 192.168.254.86:zipkin-service:9411

图 7-46　仪表盘图

（4）这时再访问http://localhost:8763/order/hello地址进行调用，然后到 zipkin server 上看有什么变化，这时可以看到服务列表里面增加了 loadbalance-service，如图 7-47 所示。

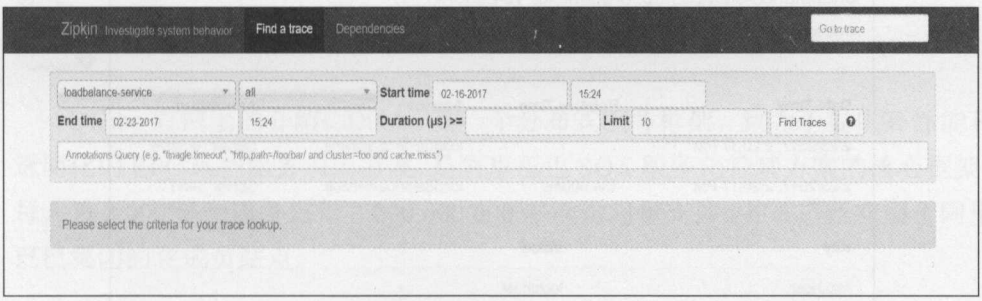


图 7-47　zipkin 界面图

（5）点击 Find Traces 来查看结果，可以看到它的完整调用链条，如图 7-48 所示。

（6）点击链接之后可以看到调用链条，如图 7-49 所示。

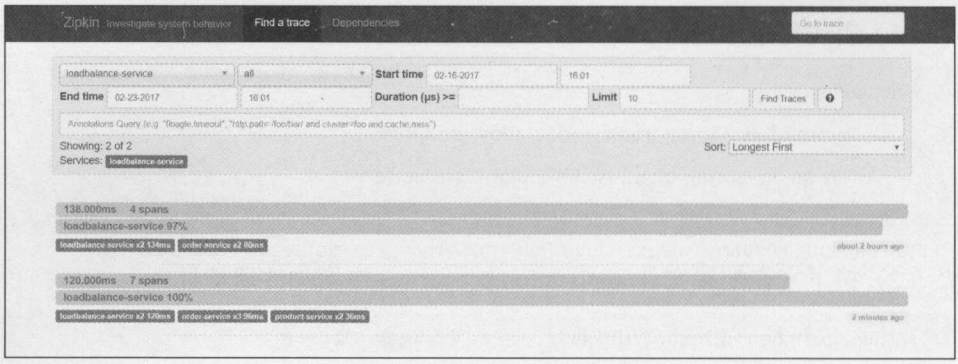


图 7-48　loadbalance 链路图

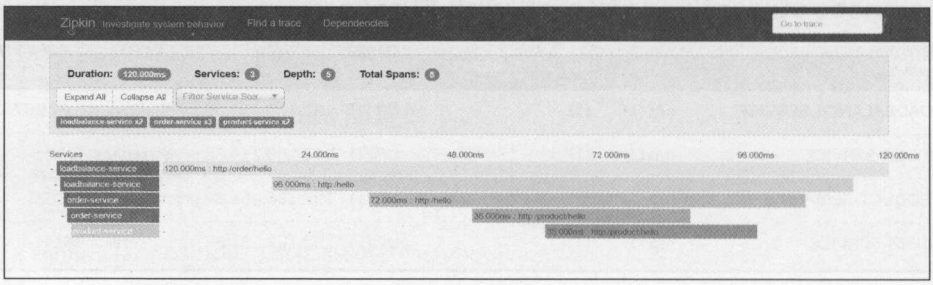


图 7-49　loadbalance 链路详图

（7）点击“查看详情”可以看到 loadbalance-service 的具体信息，如图 7-50 所示。

loadbalance-service.http:/order/hello: 134.000ms

AKA: loadbalance-service

Date Time	Relative Time	Service	Annotation
2/23/2017, 2:20:17 PM		loadbalance-service	Server Receive
2/23/2017, 2:20:17 PM	134.000ms	loadbalance-service	Server Send

Key	Value
http.host	localhost
http.method	GET
http.path	/order/hello
http.url	http://localhost:8763/order/hello

图 7-50　链路详细信息图

（8）同时可以看到依赖关系，如图 7-51 所示。

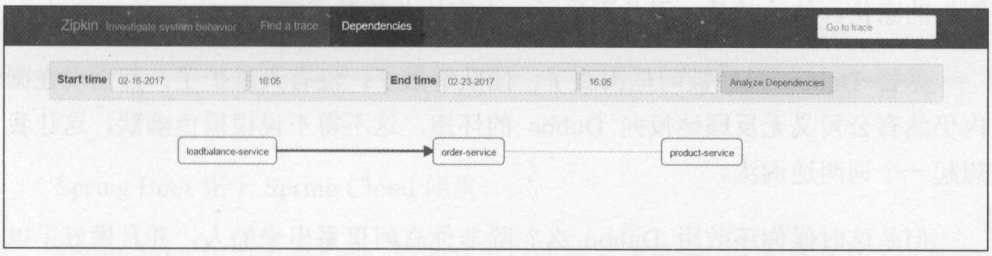


图 7-51　依赖关系图

7.10.4　小结

通过 Spring Cloud Sleuth 和 Zipkin 的结合使用，就能很方便地对微服务架构中的服务进行比较全方位的日志管理，便于出现问题时即时定位，并且可以查看哪些环节效率低，及早处理，这样能构建更为高可用的性能。

7.11　Spring Cloud 和 Dubbo 的比较

微服务在技术选型时总会讨论 Spring Cloud 和 Dubbo 之间的比较。

7.11.1　生的伟大

Dubbo 官网上的介绍：Dubbo 是一个分布式服务框架，致力于提供高性能和透明化的 RPC 远程服务调用方案，是阿里巴巴 SOA 服务化管理方案的核心框架，每天为 2 000 多个服务提供 3 000 000 000 多次访问量支持，并被广泛应用于阿里巴巴集团的各成员站点。

它是一款很成功的产品，Dubbo 在服务管理上有一套完整的解决方案，可靠性高，正式商用，实践多，国内用的公司不少。

7.11.2　死的憋屈

我花了很多时间和精力研究 Dubbo 之后，听到一个震惊的消息。

Dubbo 团队解散了，同时 Dubbo 停止更新了。Dubbo 败给了另一个叫“好舒服”的家伙。结论就是：天儿不早了，大家伙儿都散了吧。

尽管 Dubbo 已经被阿里淘汰了，团队解散了，没有人维护了，然而现在国内仍然有公司义无反顾地投向 Dubbo 的怀抱，这不得不说很黑色幽默。这让我想起一个词叫逆淘汰。

但是这时候你还敢用 Dubbo 么？除非你有阿里系出来的人，并且做好了以后自己在上面做二次开发的准备。以后全靠自己了，祝君好运。



7.11.3 没比较没伤害

Dubbo 只有服务管理这一块。

Spring Cloud 则有一套完整的框架，自带体系，不仅包含服务管理，更有其他大量丰富的套件可供选择。

相对于 Dubbo 的单一，Spring Cloud 是全套的。所以对于二者之间的比较并不是一个量级的。

在这方面 Dubbo 完全被 Spring Cloud 秒杀。

Spring Cloud 相当于拥有少林寺十八罗汉，各种技能点都有，而 Dubbo 只有老哥儿一个。



7.11.4 现状

现在国内用 Dubbo 的公司不少，在国外则是 Spring Cloud 的天下，这方面国内 IT 公司还是比较支持国货的，可能也是因为对 Spring Cloud 还不太熟悉。

如果选择 Spring Cloud，相当于拥有了十分完备的配套设施，Spring 本身背景强大，同时有众多大牌高科技公司坐镇，把自家框架开源出来做贡献，社区强大，解除了后顾之忧。

如果采用 Dubbo，就像在郊区买了一套房，配套不全，不够便利，而且不要指望着配套能不断完善，因为开发商和物业都跑了，一切都要靠自己了。

7.12 Spring Cloud 与 Spring Boot 的关系

Spring Boot 先于 Spring Cloud 问世。

Spring Boot 相当于脚手架，借助它可以快速搭建房子。它本身不具备任何功能属性，只是普通房间，没有其他功能。

房子在集成 Spring Cloud 中的组件之后，就具备了各种功能，如防火、防盗、监控。

Spring Cloud 不是强制性一次提供所有的组件，用户可以根据自己的需要进行选择，集成一个或多个组件，就像汽车可选配的全景天窗或真皮座椅。

房子想出租时就集成服务注册中心，这样房子就可以对外提供服务，相当于注册到 airbnb。

房子如果想让中介代理，那么就集成代理，中介就会帮你打理房子。

在 Spring Boot 和 Spring Cloud 集成时，不需要修改原应用的代码，只需要新增服务即可，这是非常好的实现方式，没有代码的耦合，就像乐高玩具想要具备什么功能只要和其他组件组合一下就可以，而不需要修改原组件。

有很多人分不清 Spring Boot 和 Spring Cloud 的区别。

通过代码也可以比较直观地看出 Eureka server 启动时的类，SpringBootApplication 属于 Spring Boot 包，负责启动的类，属于基础类。

服务注册和发现的 EnableEurekaClient 和 EnableEurekaServer 都是属于 Spring Cloud 下面套件的工具，还可以从 Spring Cloud 中引入其他的各种组件来共同实现微服务。

在本书 6.5 节“可定制化”里，通过 Spring Boot 定制化界面，可以看到可选择组件里面包括了 Spring Cloud 的一些组件，这样更能清楚它和 Spring Cloud 的关系，Spring Boot 相当于一个机箱，Spring Cloud 相当于是一些组件，如硬盘、CPU 等。

7.13 本章小结

Spring Cloud 提供了非常丰富的套件，覆盖了微服务架构所涉及的每一个角落，不仅有自家的组件，还集成其他第三方的优秀组件，形成了多点开花的局面，论丰富性，天下第一。

当谈到服务管理，Spring Cloud 有组件；涉及负载均衡，Spring Cloud 有组件；提到智能路由，Spring Cloud 有组件；说起服务间调用，Spring Cloud 还有组件。基本上能想到的功能，Spring Cloud 都提供了，没想到的功能，也提供了。它就像机器猫一样，从口袋里不断变出新玩意儿来满足微服务架构的需要。

然而耕良田千顷不过一日三餐，有广厦万间只睡卧榻三尺。在使用上要按需使用，不能为了使用而使用，选用的组件要符合自己的业务需求，如果引入不必要的组件只会造成过度使用从而造成混乱，就像自助餐虽然丰富，但是吃多了也会消化不良，不利于身体健康。

本书所涉及的组件虽然不是全部所有组件，但是包含了微服务架构当中必要的组件，有了这些组件就可以轻松搭建一套相对成熟的微服务架构，至于其他组件的选用就像是选配的内饰一样，就要看自己的需求了。像 Grunt 有 4000 多个插件，使用的时候也只是选用个别插件即可，不可能全部使用。



其他相关技术和工具

一个好汉三个帮，一个微服务众多工具帮。

下面介绍几个在微服务开发中比较有用的工具和技术，如数据库持续集成工具 Liquibase、服务可视化工具 Swagger、通信方式等。

实际上不仅可以在微服务架构当中使用这些工具，传统架构也可以使用这些工具，来达到事半功倍的效果。

8.1 数据库开发持续集成工具 Liquibase

Liquibase 是一个用于跟踪、管理和应用数据库变化的开源的数据库重构工具。它将所有数据库的变化（包括结构和数据）都保存在 XML、JSON 等格式的文件中，便于版本控制。

尽管 Liquibase 官网称其为数据库重构工具，但是作者认为它不仅是一个重构工具，更是一款优秀的数据库开发持续集成工具。

在微服务的使用中为了实现精益开发，引进 Liquibase 是一个很好的选择，是对软件开发中数据库版本管理的一个很好的补充，这一块也是一直被忽略的部分，以往都只关注代码的版本管理，没有关注数据库的版本管理。现在通过它来更好地完成微服务架构的开发和管理工作。



8.1.1 特性

（1）支持多种主流数据库，包括 Oracle、SQL Server、DB2、MySQL、Sybase、PostgreSQL 等，在微服务的数据库选型上，可选择范围特别广。

（2）支持多种格式存储数据库脚本，如 YML、JSON、SQL、XML 等，以作者和 ID 唯一标识一个变化（ChangSet），支持数据库脚本的合并，支持开发人员协同工作。下面来看一下 YML、JSON 和 SQL 格式样例。

YML 格式如下。

```
databaseChangeLog:
  - preConditions:
    - runningAs:
        username: Liquibase
  - changeSet:
      id: 1
      author: nvoxland
      changes:
        - createTable:
            tableName: person
            columns:
              - column:
                  name: id
                  type: int
                  autoIncrement: true
                  constraints:
                    primaryKey: true
                    nullable: false
              - column:
                  name: firstname
                  type: varchar(50)
              - column:
                  name: lastname
                  type: varchar(50)
                  constraints:
                    nullable: false
              - column:
```

```
                name: state
                type: char(2)
```

JSON 格式如下。

```
{
    "databaseChangeLog": [
        {
            "preConditions": [
                {
                    "runningAs": {
                        "username": "Liquibase"
                    }
                }
            ]
        },
        {
            "changeSet": {
                "id": "1",
                "author": "nvoxland",
                "changes": [
                    {
                        "createTable": {
                            "tableName": "person",
                            "columns": [
                                {
                                    "column": {
                                        "name": "id",
                                        "type": "int",
                                        "autoIncrement": true,
                                        "constraints": {
                                            "primaryKey": true,
                                            "nullable": false
                                        },
                                    }
                                },
                                {
                                    "column": {
```

```
                    "name": "firstname",
                    "type": "varchar(50)"
                }
            },
            {
                "column": {
                    "name": "lastname",
                    "type": "varchar(50)",
                    "constraints": {
                        "nullable": false
                    },
                }
            },
            {
                "column": {
                    "name": "state",
                    "type": "char(2)"
                }
            }
        ]
    }
  }
 ]
 }
}
```

SQL 格式如下。

```
--Liquibase formatted sql
--changeset nvoxland:1
create table test1 (
    id int primary key,
    name varchar(255)
);
--rollback drop table test1;
--changeset nvoxland:2
```

```
insert into test1 (id, name) values (1, 'name 1');
insert into test1 (id, name) values (2, 'name 2');
--changeset nvoxland:3 dbms:oracle
create sequence seq_test;
```

（3）支持定义上下文，通过定义上下文将 context 属性设置成不同值，可以对应到不同环境，如测试或者是本地环境，在执行过程中通过指定上下文，就可以针对不同的环境执行不同的脚本，方便进行测试，而这部分标注上下文的 changeSet 在不指定 context 时将不会在生产环境中执行，会自动跳过。

```
<changeSet id="2" author="bob" context="test">
    <insert tableName="news">
        <column name="id" value="1"/>
        <column name="title" value="Liquibase 0.8 Released"/>
    </insert>
    <insert tableName="news">
        <column name="id" value="2"/>
        <column name="title" value="Liquibase 0.9 Released"/>
    </insert>
</changeSet>
```

（4）支持代码分支与合并，像 SVN 等版本工具一样，支持对于数据库脚本的分支与合并。还支持回滚功能，可按标签（tag）回滚到之前的版本，类似于 SVN 的版本号，回退代码的版本。

（5）自动生成运行结果便于 DBA review。

（6）集群安全的数据库升级。

（7）生成数据库变更文档，将所有变更输出文档，成为版本历史，同时所有数据库的修改历史，会自动保存到系统创建的 databasechangelog 表中，在下一次数据库升级时会自动跳过已经执行的脚本。

（8）支持数据库比较功能，比较两个数据库之间的差异，并将结果进行保存，根据差异可以轻松部署或升级数据库。

（9）支持多种方式运行（maven,ant,command line 等）。

（10）可以对离线数据库进行更新。

（11）提供了 IDE 插件，如 Eclipse 插件。



8.1.2 使用

使用过程如图 8-1 所示。

（1）下载 Liquibase。

（2）创建 changelog 文件，文件格式自己选择。

（3）然后增加 changeset，每一个 changeset 就是一个标明了作者和 ID 的 SQL 脚本。

（4）执行数据库更新。

（5）提交文件到版本控制中心。

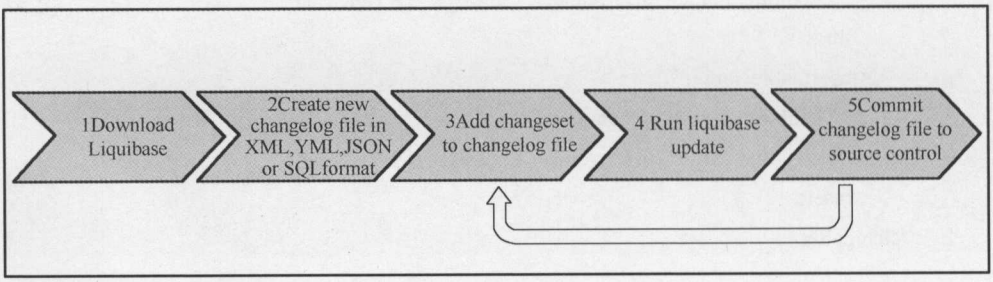


图 8-1　Liquibase 使用过程图

在工程中使用的步骤如下。

（1）在 pom.xml 中增加 Maven 插件，同时配置属性文件的路径和名称。

```
<plugin>
        <groupId>org.liquibase</groupId>
        <artifactId>liquibase-maven-plugin</artifactId>
        <version>3.0.5</version>
        <configuration>
          <propertyFile>src/main/resources/liquibase.properties</propertyFile>
        </configuration>
        <executions>
          <execution>
            <phase>process-resources</phase>
            <goals>
              <goal>update</goal>
```

```
            </goals>
          </execution>
        </executions>
      </plugin>
```

（2）在上面配置的目录 src/main/resources/下增加 liquibase.properties 配置文件。

指定 changeLogFile 是需要执行的脚本文件。

```
contexts: ${liquibase.contexts}
changeLogFile: config/liquibase/changelog/db.changelog.xml
driver: com.mysql.jdbc.Driver
url: jdbc:mysql://localhost:3306/Liquibase
username: xxx
password: xxx
verbose: true
dropFirst: false
```

（3）数据库脚本文件 db.changelog.xml 内容如下。

```
<databaseChangeLog xmlns="http://www.Liquibase.org/xml/ns/dbchangelog" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.Liquibase.org/ xml/ns/dbchangelog http://www.Liquibase.org/xml/ns/dbchangelog/dbchangelog-3.0.xsd">
    <changeSet author="phil" id="001">
        <createTable tableName="TEST_LIQUIBASE">
            <column autoIncrement="true" name="id" type="INT(10)">
                <constraints primaryKey="true"/>
            </column>
            <column name="firstname" type="VARCHAR(50)"/>
            <column name="lastname" type="VARCHAR(50)">
                <constraints nullable="false"/>
            </column>
            <column name="state" type="VARCHAR(2)"/>
        </createTable>
    </changeSet>
</databaseChangeLog>
```

脚本中定义作者、主键 ID 的值，这些都会保存到系统表里面，这是一个普通创建表的语句，只是通过 XML 形式存在。

（4）通过 Mvn 命令执行更新命令。

```
C:\Users\xxx\workspace\jzb-userservice>mvn liquibase:update
[INFO] Scanning for projects...
[WARNING]
[WARNING] Some problems were encountered while building the effective model for userservice:jar:0.0.1-SNAPSHOT
[WARNING] 'dependencies.dependency.(groupId:artifactId:type:classifier)' must be unique: org.springframework.boot:spring-boot-starter-security:jar -
ration of version (?) @ line 458, column 21
[WARNING] 'build.plugins.plugin.version' for org.apache.maven.plugins:maven-surefire-report-plugin is missing. @ line 790, column 21
[WARNING]
[WARNING] It is highly recommended to fix these problems because they threaten the stability of your build.
[WARNING]
[WARNING] For this reason, future Maven versions might no longer support building such malformed projects.
[WARNING]
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building userservice 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- Liquibase-maven-plugin:3.0.5:update (default-cli) @ userservice ---
[INFO] ------------------------------------------------------------------------
[INFO] Parsing Liquibase Properties File
[INFO]     File: src/main/resources/Liquibase.properties
[INFO] ------------------------------------------------------------------------
[INFO] Executing on Database: jdbc:mysql://localhost:3306/Liquibase
INFO 2/3/17 9:43 PM:Liquibase: null: null: Successfully acquired change log lock
INFO 2/3/17 9:43 PM:Liquibase: null: null: Creating database history table with name: DATABASECHANGELOG
INFO 2/3/17 9:43 PM:Liquibase: null: null: Reading from DATABASECHANGELOG
INFO 2/3/17 9:43 PM:Liquibase: null: null: Reading from DATABASECHANGELOG
INFO 2/3/17 9:43 PM:Liquibase: config/Liquibase/db.changelog.xml: config/Liquibase/db.changelog.xml::001::phil: Table TEST_LIQUIBASE created
INFO 2/3/17 9:43 PM:Liquibase: config/Liquibase/db.changelog.xml: config/Liquibase/
```

```
db.changelog.xml::001::phil: ChangeSet config/Liquibase/db.change
    l ran successfully in 54ms
    INFO 2/3/17 9:43 PM:Liquibase: config/Liquibase/db.changelog.xml: null: Successfully released change log lock
    [INFO] ------------------------------------------------------------------------
    [INFO]
    [INFO] ------------------------------------------------------------------------
    [INFO] BUILD SUCCESS
    [INFO] ------------------------------------------------------------------------
    [INFO] Total time: 11.330 s
    [INFO] Finished at: 2017-02-03T21:43:05+08:00
    [INFO] Final Memory: 25M/275M
    [INFO] ------------------------------------------------------------------------
```

构建成功之后可以看到数据库里多了三张表，一张是刚才 SQL 语句创建的测试表，另外两个是系统表，是用来记录数据库版本的。

```
    mysql> use liquibase;
    Database changed
    mysql> show tables;
    +----------------------+
    | Tables_in_Liquibase  |
    +----------------------+
    | databasechangelog    |
    | databasechangeloglock |
    | test_Liquibase       |
    +----------------------+
    3 rows in set (0.00 sec)
```

系统表里会详细记录每次更新的操作信息，如下面结果所示，记录了刚才的操作。

```
    # ID, AUTHOR, FILENAME, DATEEXECUTED, ORDEREXECUTED, EXECTYPE, MD5SUM, DESCRIPTION, COMMENTS, TAG, LIQUIBASE
    '001', 'phil', 'config/liquibase/db.changelog.xml', '2017-02-03 21:43:05', '1', 'EXECUTED', '7:feb0db0a637542795156174708387168', 'Create Table', '', NULL, '3.0.5'
```

另一个脚本文件 db-changelog-001.xml 的内容如下。

```
    <?xml version="1.0" encoding="UTF-8"?>
```

```
    <databaseChangeLog xmlns="http://www.Liquibase.org/xml/ns/dbchangelog"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.Liquibase.org/xml/ns/dbchangelog
http://www.Liquibase.org/xml/ns/dbchangelog/dbchangelog-2.0.xsd">
        <changeSet author="fresh" id="123456789-0" runOnChange="true">
            <createTable tableName="TESTUSER">
                <column name="USSTID" type="VARCHAR(10)" />
                <column name="STATUS" type="VARCHAR(10)" />
                <column name="LEGALENTITY" type="VARCHAR(10)" />
                <column name="VALIDTODATE" type="DATE" />
                <column name="VALIDFROMDATE" type="DATE" />
            </createTable>
            </changeSet>
            <changeSet author="phil" id="123456789-1" runOnChange="true">
            <addPrimaryKey columnNames="USSTID" constraintName="PK_USSTID" tableName=
"TESTUSER" />
            <addNotNullConstraint columnName="VALIDTODATE" columnDataType="DATE"
tableName= "TESTUSER" />
            <addNotNullConstraint columnName="VALIDFROMDATE" columnDataType="DATE"
tableName= "TESTUSER" />
            <addNotNullConstraint columnName="LEGALENTITY" columnDataType="VARCHAR(10)"
tableName= "TESTUSER" />
        </changeSet>

    </databaseChangeLog>
```

（5）再次执行 C:\Users\xxx\workspace\jzb-userservice>mvn liquibase:update。

```
    [INFO] Scanning for projects...
    [WARNING]
    [WARNING] Some problems were encountered while building the effective model for
userservice:jar:0.0.1-SNAPSHOT
    [WARNING] 'dependencies.dependency.(groupId:artifactId:type:classifier)' must be unique:
org.springframework.boot:spring-boot-starter-security:jar -> duplicate decla
    ration of version (?) @ line 458, column 21
    [WARNING] 'build.plugins.plugin.version' for org.apache.maven.plugins:maven-surefire-report-
plugin is missing. @ line 790, column 21
    [WARNING]
```

```
[WARNING] It is highly recommended to fix these problems because they threaten the
stability of your build.
[WARNING]
[WARNING] For this reason, future Maven versions might no longer support building such
malformed projects.
[WARNING]
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building userservice 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- Liquibase-maven-plugin:3.0.5:update (default-cli) @ userservice ---
[INFO] ------------------------------------------------------------------------
[INFO] Parsing Liquibase Properties File
[INFO]     File: src/main/resources/Liquibase.properties
[INFO] ------------------------------------------------------------------------
[INFO] Executing on Database: jdbc:mysql://localhost:3306/Liquibase
INFO 2/3/17 9:58 PM:Liquibase: null: null: Successfully acquired change log lock
INFO 2/3/17 9:58 PM:Liquibase: null: null: config/Liquibase/changelog/db-changelog-
001.xml is using schema version 2.0 rather than version 3.0
INFO 2/3/17 9:58 PM:Liquibase: null: null: Reading from DATABASECHANGELOG
INFO 2/3/17 9:58 PM:Liquibase: null: null: Reading from DATABASECHANGELOG
INFO 2/3/17 9:58 PM:Liquibase: config/Liquibase/changelog/db-changelog-001.xml:
config/Liquibase/changelog/db-changelog-001.xml::123456789-1::phil: Primary key added
    to TESTUSER (USSTID)
INFO 2/3/17 9:58 PM:Liquibase: config/Liquibase/changelog/db-changelog-001.xml:
config/Liquibase/changelog/db-changelog-001.xml::123456789-1::phil: Null constraint h
    as been added to TESTUSER.VALIDTODATE
INFO 2/3/17 9:58 PM:Liquibase: config/Liquibase/changelog/db-changelog-001.xml:
config/Liquibase/changelog/db-changelog-001.xml::123456789-1::phil: Null constraint h
    as been added to TESTUSER.VALIDFROMDATE
INFO 2/3/17 9:58 PM:Liquibase: config/Liquibase/changelog/db-changelog-001.xml:
config/Liquibase/changelog/db-changelog-001.xml::123456789-1::phil: Null constraint h
    as been added to TESTUSER.LEGALENTITY
INFO 2/3/17 9:58 PM:Liquibase: config/Liquibase/changelog/db-changelog-001.xml:
config/Liquibase/changelog/db-changelog-001.xml::123456789-1::phil: ChangeSet config/
    Liquibase/changelog/db-changelog-001.xml::123456789-1::phil ran successfully in 650ms
```

```
INFO 2/3/17 9:58 PM:Liquibase: config/Liquibase/changelog/db-changelog-001.xml: null:
Successfully released change log lock
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 11.519 s
[INFO] Finished at: 2017-02-03T21:58:32+08:00
[INFO] Final Memory: 26M/293M
[INFO] ------------------------------------------------------------------------
```

成功之后可以看到系统表里多了一条操作记录，如图 8-2 所示。

id	AUTHOR	FILENAME	ORDER	DESCRIPTION	TAG
1	phil	config/liquibase/db.changelog.xml	1	Create Table	NULL
123456789-0	fresh	config/liquibase/changelog/db-changelog-001.xml	2	Create Table	NULL
123456789-1	phil	config/liquibase/changelog/db-changelog-001.xml	3	Adds creates a primary	NULL

图 8-2　变更表图

（6）这时如果像 SVN 一样打个版本号标记则执行如下命令。

mvn liquibase:tag -Dliquibase.tag=tagV1

C:\Users\xxx\workspace\jzb-userservice>mvn Liquibase:tag -DLiquibase.tag=tagV1

```
[INFO] Scanning for projects...
[WARNING]
[WARNING] Some problems were encountered while building the effective model for
jzb-userservic
[WARNING] 'dependencies.dependency.(groupId:artifactId:type:classifier)' must be unique:
org.springframework.boot:spri
ration of version (?) @ line 458, column 21
[WARNING] 'build.plugins.plugin.version' for org.apache.maven.plugins:maven-surefire-
report-plugin is missing. @ line
[WARNING]
[WARNING] It is highly recommended to fix these problems because they threaten the
stability of your build.
[WARNING]
[WARNING] For this reason, future Maven versions might no longer support building such
malformed projects.
```

```
[WARNING]
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building jzb-userservice 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- Liquibase-maven-plugin:3.0.5:tag (default-cli) @ jzb-userservice ---
[INFO] ------------------------------------------------------------------------
[INFO] Parsing Liquibase Properties File
[INFO]     File: src/main/resources/Liquibase.properties
[INFO]     'changeLogFile' in properties file is not being used by this task.
[INFO]     'dropFirst' in properties file is not being used by this task.
[INFO]     'contexts' in properties file is not being used by this task.
[INFO] ------------------------------------------------------------------------
[INFO] Executing on Database: jdbc:mysql://localhost:3306/Liquibase
INFO 2/3/17 10:11 PM:Liquibase: null: null: Successfully acquired change log lock
INFO 2/3/17 10:11 PM:Liquibase: null: null: Reading from DATABASECHANGELOG
INFO 2/3/17 10:11 PM:Liquibase: null: null: Successfully released change log lock
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 7.968 s
[INFO] Finished at: 2017-02-03T22:11:12+08:00
[INFO] Final Memory: 24M/279M
[INFO] ------------------------------------------------------------------------
```

执行成功之后，查看数据库记录发现标记已经打到最后一次操作上面了，如图 8-3 所示。

id	AUTHOR	FILENAME	ORDER	DESCRIPTION	TAG
1	phil	config/liquibase/db.changelog.xml	1	Create Table	NULL
123456789-0	fresh	config/liquibase/changelog/db-changelog-001.xml	2	Create Table	NULL
123456789-1	phil	config/liquibase/changelog/db-changelog-001.xml	3	Adds creates a primary	tagV1

图 8-3　变更表图

再执行更新操作之后，会发现 changelog 里面已经增加了记录，如图 8-4 所示。

id	AUTHOR	FILENAME	ORDER	DESCRIPTION	TAG
1	phil	config/liquibase/db.changelog.xml	1	Create Table	NULL
123456789-0	fresh	config/liquibase/changelog/db-changelog-001.xml	2	Create Table	NULL
123456789-1	phil	config/liquibase/changelog/db-changelog-001.xml	3	Adds creates a primary	tagV1
123456789-2	Feb	config/liquibase/changelog/db-changelog-002.xml	4	Create Table	NULL

图 8-4　变更表图

（7）这时使用回滚命令，回退到之前的 TAG1 版本。

```
mvn liquibase:rollback -Dliquibase.rollbackTag=tagV1
```

使用这个命令之后整个数据库会回退到这个标记时的版本，最后创建的表已经不存在了，同时 changelog 里面对应的这条记录也已经被清空。

8.2　Swagger

Swagger is a powerful open source framework backed by a large ecosystem of tools that helps you design, build, document, and consume your RESTful APIs.

通过 Swagger 官网上的说明可以看出它是一个开源框架，可以帮助使用者设计、构建、文档化，使用 RESTful 服务 API。

说白了就是 Swagger 可以生产、显示和测试 RESTful 服务。



8.2.1　为什么使用 Swagger

微服务是面向服务的架构，服务数量多，每个服务又包括一系列 API，服务调用者需要一个使用说明书，知道这些服务都有什么功能，以及如何获取这些功能。

那么如何将这些 API 暴露出来，让调用者清楚地知道呢？

答案是通过接口可视化工具。

Swagger 的好处在于接口可视化，在线文档和 API 始终同步，以及可以对接口进行测试。

接口可视化让每个服务都仿佛自带使用说明书，功能透明，外部消费者能很

清楚地知道这个微服务具备哪些功能，通过传递参数，会得到什么功能。消费者只要按照使用说明书进行操作就可以了。

就像通过自动售货机购物一样，使用非常方便，一眼就能看出机器提供的所有货品，也就是全部的服务，以及如何获取这些服务都有特别明确的说明，十分便捷。

传统方式一般通过文档来对服务进行定义，使用者根据文档进行使用。

但是这时存在一个问题，因为 API 会调整，对应的文档也必须同步更新，两者要保持高度的同步，否则使用者所使用的就是过期的使用说明书，和实际不匹配。

而因为传统上文档和代码是分开的，开发人员开发完代码，做完测试，打包发布之后，很容易就把文档给忘记了，或者更新了文档但是忘记提交到系统上，这样服务调用方还是无法获取最新的文档。

一旦出现两者版本不统一的情况，就会让其他使用者已经辛苦开发出来的代码毫无用处，这时不是简单修改一下就可以，可能会造成大范围的修改，工作量非常大。

普通文档已经无法满足需要，更方便的方式是使用服务可视化工具。

Swagger 是服务可视化工具，然而功能不仅限于此，包括 Swagger UI、Swagger Editor、Swagger Codegen 等。它可以同步更新文档，相当于文档自动更新。而且可以通过浏览器查看应用的服务列表，并且直接进行测试。

Swagger 的一大优势是将传统上互相独立的代码和文档进行了动态的结合，开发人员只需要在同一地方修改代码和注释即可以保证文档的更新，并且所有人都可以看到最新版本。

Swagger 的原理就是通过在接口上增加注解描述，来达到接口可视化的效果，同时只要将更新之后的代码打包发布，接口和文档就会同步更新，解决了一致性的问题。



8.2.2 如何使用

（1）pom.xml 增加依赖。

```
<dependency>
```

```
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.0.3</version>
</dependency>
<dependency>
    <groupId>com.mangofactory</groupId>
    <artifactId>swagger-springmvc</artifactId>
    <version>0.9.5</version>
</dependency>
    <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.0.3</version>
</dependency>
```

（2）application.yml 增加。

```
swagger:
    title: xxservice API
    description: xxservice applications and beyond!
    version: 0.0.1
    termsOfServiceUrl: http://jhipster.github.io/
    contact:
    license: Apache 2.0
    licenseUrl: http://www.apache.org/licenses/LICENSE-2.0.html
```

（3）新建类 SwaggerConfiguration 用来配置 Swagger。

```
@Configuration
@EnableSwagger2
@ComponentScan("com.jzb.web.rest")
public class SwaggerConfiguration implements EnvironmentAware {
    private final Logger log = LoggerFactory
            .getLogger(SwaggerConfiguration.class);
    public static final String DEFAULT_INCLUDE_PATTERN = "/api/.*";
    private RelaxedPropertyResolver propertyResolver;
    /**
     * Set Environment
```

```
     */
    @Override
    public void setEnvironment(Environment environment) {
        this.propertyResolver = new RelaxedPropertyResolver(environment,
                "swagger.");
    }
    /**
     * Swagger Springfox configuration.
     */
    @Bean
    public Docket swaggerSpringfoxDocket() {
        log.debug("Starting Swagger");
        StopWatch watch = new StopWatch();
        watch.start();
        Docket docket = new Docket(DocumentationType.SWAGGER_2)
                .apiInfo(apiInfo())
                .genericModelSubstitutes(ResponseEntity.class)
                .forCodeGeneration(true)
                .genericModelSubstitutes(ResponseEntity.class)
                .directModelSubstitute(org.joda.time.LocalDate.class,
                        String.class)
                .directModelSubstitute(org.joda.time.LocalDateTime.class,
                        Date.class)
                .directModelSubstitute(org.joda.time.DateTime.class, Date.class)
                .directModelSubstitute(java.time.LocalDate.class, String.class)
                .directModelSubstitute(java.time.ZonedDateTime.class,
                        Date.class)
                .directModelSubstitute(java.time.LocalDateTime.class,
                        Date.class).select()
                .paths(regex(DEFAULT_INCLUDE_PATTERN)).build();
        watch.stop();
        log.debug("Started Swagger in {} ms", watch.getTotalTimeMillis());
        return docket;
```

```
        }
        /**
         * API Info as it appears on the swagger-ui page.
         */
        private ApiInfo apiInfo() {
            return new ApiInfo(propertyResolver.getProperty("title"),
                    propertyResolver.getProperty("description"),
                    propertyResolver.getProperty("version"),
                    propertyResolver.getProperty("termsOfServiceUrl"),
                    propertyResolver.getProperty("contact"),
                    propertyResolver.getProperty("license"),
                    propertyResolver.getProperty("licenseUrl"));
        }
    }
```

（4）在 Rest 接口代码中通过注解的方式使用 swagger。

代码如下。

```
    @RestController
    @RequestMapping(value = "/api/rest")
    @Api(value = "DiggermachineOverview", description = "Diggermachine Overview Information ")
    public class DiggermachineOverviewWebService {

        @ResponseBody
        @RequestMapping(value = "/hello", method = RequestMethod.GET)
        @ApiOperation(nickname = "swagger-ui world", value = "SwaggerUI 的世界", notes = "测试 swagger ui")
        public String hello(@ApiParam(value = "接口测试参数") @RequestParam String apiParam) {
            return "Hello , " + apiParam;
        }

    }
```

（5）然后通过浏览器访问以下网址。

http://localhost:8374/diggerMachineService/swagger-ui.html

可以看到 API 列表，并且可以进行测试，如图 8-5 所示。

（6）点击 try it out 之后就可以看到测试结果，如图 8-6 所示。

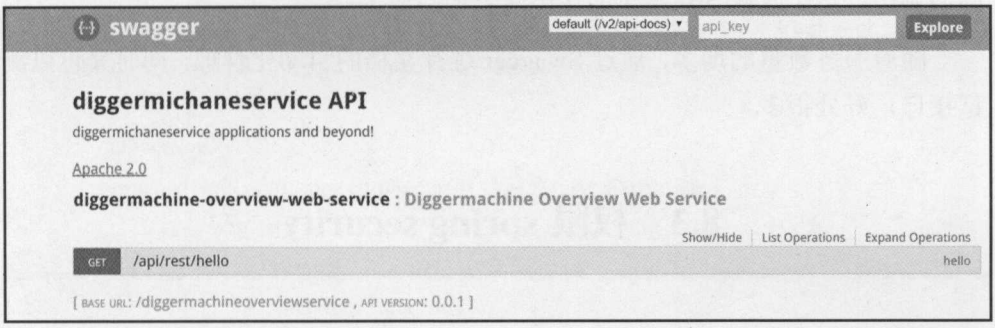


图 8-5　swagger 图

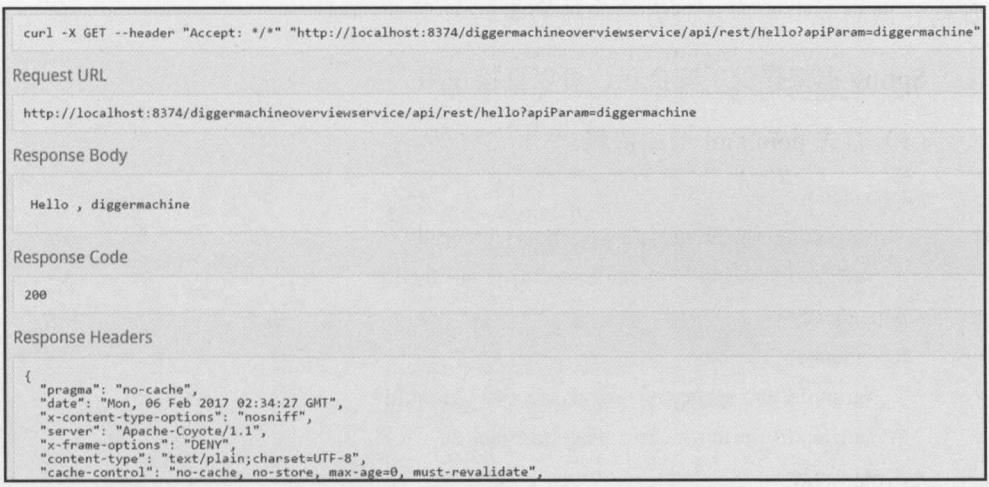


图 8-6　测试结果图



8.2.3　小结

在以往前后端开发分离的模式下，会带来两者沟通的工作量，围绕着 API 接口的定义和实现来进行。通过服务化 API 进行契约式声明，双方就要遵循这个契约进行开发，但是 API 调整的情况在所难免，这时就特别需要一个纽带让双方紧密联系起来。

Swagger 一直都在充当前后端交流的重要桥梁，是因为其方便快捷，易于理解。

在微服务架构中，不再仅限于前后端之间的服务调用，更多的是后端服务之间的调用，这个时候 Swagger 的作用就更加凸显了。

随着服务数量的增多，通过 Swagger 进行文档的自动化管理，同时又可以测试接口，好处很多。

8.3 权限 spring security

所有系统都要涉及权限，在微服务架构中如何鉴权同样是非常重要的，那么如何在微服务中实现权限方面的校验呢？

Spring 框架提供了安全包，可以直接使用。

（1）首先 pom.xml 增加依赖。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-ldap</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <version>${spring.security-test.version}</version>
    <scope>test</scope>
</dependency>
```

（2）扩展 WebSecurityConfigurerAdapter 这个类，然后对 URL 进行权限设置，哪些是不需要权限的就设置成 permitAll，需要权限的就设置成 authenticated，需要管理员权限的就设置成 ADMIN。

代码如下。

```
@Configuration
@EnableWebSecurity
@EnableConfigurationProperties
@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true)
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
            http.csrf()
                .disable()
                .exceptionHandling()
                .accessDeniedHandler(new AccessDeniedHandler() {
                    @Override
                    public void handle(HttpServletRequest request,
                                       HttpServletResponse response,
                                       AccessDeniedException accessDeniedException)
                                       throws IOException, ServletException {

                        response.sendError(HttpServletResponse.SC_FORBIDDEN,
                                           "Access Denied");
                    }
                }).authenticationEntryPoint(authenticationEntryPoint).and()
                .addFilter(requestBasicAuthenticationFilter())
                .authorizeRequests().antMatchers("/api/**"). authenticated ()

                .antMatchers("/v2/api-docs/**").permitAll()
                .antMatchers("/configuration/security")
                .hasAuthority(AuthoritiesConstants.ADMIN)
                .antMatchers("/configuration/ui")
                .hasAuthority(AuthoritiesConstants.ADMIN)
                .antMatchers("/swagger-ui**").permitAll()
                .antMatchers("/protected/**").authenticated();
    }
}
```

（3）实现 UserDetailsService 这个接口，重写 loadUserByUsername 方法。

代码如下。

```
@Override
public UserDetails loadUserByUsername(final String username) {
        XXXUserDetails ud = new XXXUserDetails(username);
    try {
        List<XXXRole> roles = userRestService.getRolesForUser(roleContext, username);
        for (XXXRole role : roles) {
            ud.addAuthority(role.getRoleName());
        }

        return ud;
    } catch (Exception e) {
        throw new UsernameNotFoundException(String.format("User '%s' not found in
user service", username), e);
    }
}
```

（4）最后通过在 rest 接口方法上增加注解来达到权限管理的目的。

```
@PreAuthorize("hasRole('XXXUSERADMIN')")
```

8.4 微服务架构的通信方式

在微服务架构中，由于分布式的特点，服务数量多，服务之间如何通信是一个首先要面对的问题，通信方式的选择，关乎各个服务功能是否正常以及系统整体的稳定性，事关重大。

百团大战时，很多个独立的师团分布在不同的地界，就像分布式系统一样，这些师团之间只有通过及时、稳定的通信才能保证在战前制订好的作战计划可以在作战时得到很好的贯彻和执行，并且能够根据势态的发展快速应对。如果一旦出现通信不畅、联系不上的情况，那么受影响的不仅仅是个别师团的作战情况，可能是整个战役的走向，甚至是国家的历史进程，影响范围特别大。

微服务对于通信方式的选择标准也要和整体架构风格保持一致，那就是轻

量、低耦合。如果是重量级的通信方式显然与微服务架构风格格格不入，无法支撑整个架构的正常运行，只会拖累系统的整体性能。

下面讨论一下通信的两种主要方式：同步和异步。

两者的区别是同步方式是实时反馈结果，异步方式是非实时反馈结果。

同步和异步的一个判断标准是看请求方是否需要等待结果的返回。

比如，办理一些证件时，有些证件是立等可取，像暂住证，这种可以当成同步的；有些证当时办不下来，会先给一个凭据，约定时间再取或通过快递配送，不用办理人一直在那里等待，像港澳通行证，这种相当于异步。

在现实生活中同步方式往往要比异步方式快，而在系统当中异步并不一定比同步慢，同步就像一个工人一次做一支枪，而异步就像批量式生产零件、批量组装。所以同样做一百支枪的时间，异步可能比同步快。

同步和异步之间就像文无第一，没有绝对的孰优孰劣，要采用哪种方式需要综合考虑，有些情况下采用异步的方式可以缓解系统的压力，防止阻塞的发生，更利于系统的稳定，也是一种不错的方案。一个系统当中经常同步和异步方式共存，结合起来使用会取得不错的效果。



8.4.1 同步

日常生活中人们经常会排队，最难的排队如春节买火车票、去医院挂号等。这些都是同步、阻塞式的处理方式，只能 one by one，每完成一个才能轮到下一个，这样效率比较低。

系统中如果请求数量巨大容易造成排队并发生阻塞，严重影响系统的正常运行。

同步的实现方式有很多种，如 Rest、SOAP、RPC 等，其中 SOAP 是面向方法的，Rest 是面向资源的，RPC 是面向接口的。

传统上 WebService 比较常见的实现方式，无论是 CXF 还是 AXIS 系列都只是解决了一时的问题，使用上还是有诸多不便。

曾经的 SOAP、RPC 已经逐渐被 Rest 替代。亚马逊和谷歌已经将系统里原来的 SOAP 方式调用改成了 Rest 方式调用。

说到 RPC，不得不提分布式框架 Dubbo，它就是采用 RPC 来通信的，不支持 Rest 方式，然而很多人迫切要求 Dubbo 具备 Rest 功能，然后，当当在自己的 DubboX 上开发了支持 Rest 的版本。

在微服务架构中 Rest 是更好的选择，在其他业务场景下，RPC 也有可取之处，这里不介绍太多 SOAP 和 RPC 的内容，读者可以自行了解，我们这里只谈 Rest。

Rest 是一种架构风格，是一套规范，面向资源的服务，不是指一种技术。它的实现是通过传统的 HTTP+json/xml 方式完成的。

互联网上一切皆为资源，那么对于资源的正确打开姿势就是面向资源的服务。

在传统软件开发中，始终遵循的一条原则是面向接口编程，现在在这个原则之上，进行了包装，整个系统采用面向服务架构，服务下面再通过调用接口的方式实现具体功能。

Rest 模式的 Web 服务与复杂的 SOAP 和 RPC 相比较更简洁，如图 8-7 所示。

越来越多的 Web 服务开始采用 Rest 风格设计和实现。例如，Amazon.com 提供接近 Rest 风格的 Web 服务进行图书查找，雅虎提供的 Web 服务也是 Rest 风格的。

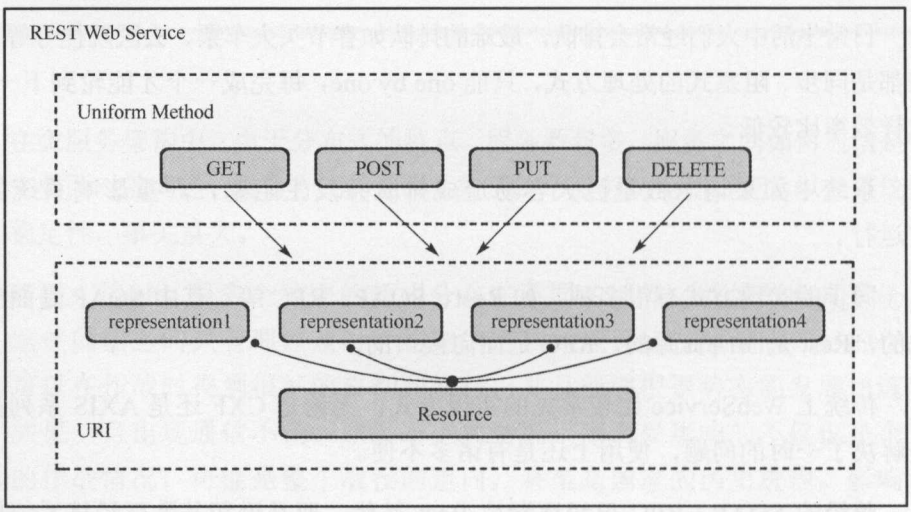


图 8-7　Rest 示意图

Rest 的主要特点：

（1）每个资源都有一个 ID；

（2）连接资源在一起；

（3）使用标准方法；

（4）资源多重表述；

（5）无状态通信。

Rest 的优势如下。

➢ 易开发、易维护

只需要几行代码就可以开发出一个 Rest 服务。

使用 RPC 的话，需要调用接口方法，代码上有渗透，虽然调用远程和调本地接口差不多，然而还是有代码的耦合关系，不利于维护和替换。

AXIS、CXF 都会根据 WSDL 文件生成客户端代码，调用时代码耦合很多这种第三方客户端代码，接口修改之后，又要重新生成客户端代码，非常麻烦，同时系统要集成一大堆代码，对平台和语言有依赖，耦合更加严重。

而 Rest 非常轻量，没有任何代码的侵入、耦合，非常开放，不依赖于平台和语言。

➢ 非常直观，所见即所得

Rest 直接通过 URL 获取指定服务，只要将接口定义清楚，那么所有的服务信息一清二楚。接口的定义都是标准的，让每个人对它的理解都是一样的，避免了产生误解的现象。

同时 Rest 的无状态性以及对于缓存的利用都让它在微服务架构中作为通信方式更加游刃有余。

Rest 示例：通过@RestController 开启 Rest 功能。

@RequestMapping(value = "/api/rest")是请求地址，在方法上可以看到请求类型是 GET，格式是 JSON，方法的请求路径是/cars/car/{carNo}，这样就对外暴露了一个根据车辆编号获取车详情的服务，请求地址就是：http://IP:port/api/rest/cars/car/001。这样就可以获取汽车详情信息了。

Rest 的请求类型共有四种：

GET 获取资源；

PUT 更新资源；

DELETE 删除资源；

POST 创建资源。

Rest 的格式包括 JSON 和 XML。

```
@RequestMapping(value = "/api/rest")
@RestController
public class CarOverviewWebService {
    @RequestMapping(method = RequestMethod.GET, value = "/cars/car/{carNo}",
produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<CarDetailDataResponseDTO> getCarByCarNo(
                                @PathVariable(value = "carNo") String carNo
                ) {
                // Response
            return new ResponseEntity<CarDetailDataResponseDTO>(
                    carDetailDataResponse, HttpStatus.OK);

                }

}
```



8.4.2 异步

在某些业务场景下，采用异步的方式，可以立即给用户一个响应，然后放到消息队列里再去逐个处理，最终的结果都是一样的。

使用异步消息的好处：首先是解耦，通过消息队列的使用，生产者和消费者都面向接口编程，这样后期无论是扩展还是替换都很容易。

- better late than never

系统难免会遇到个别组件不可用的情况，这时如果采用异步消息队列，

那么任务在管道中保存，如果没有执行成功，可以换个姿势重来一次，这样当组件恢复正常时就可以重新执行，不会丢失。这种允许重试和延时处理的能力，保证了系统的稳定性，客户体验更好，毕竟稍后得到结果要远比没有结果好很多。

- 吞吐能力强

相比于同步的处理方式，消息队列无疑提供了更为强大的处理能力，消息队列本身就是一种缓冲的机制，用来应对流量洪峰，极大了缓解后端的压力。同步的方式只能在前端采取排队、限流的方式来缓解压力。比较常见的就是早高峰时各个地铁站外的蛇形通道限流，如果不限流，人流直接涌到站台就会很危险，就如同应对系统同时应对大量请求。

异步的方式就像先把乘客放到传送带上，有很大的缓冲空间，用户体验更好。

消息队列还有一些好处，如便于理解、顺序执行、保证送达等。

因为消息队列的这么多好处，现在大量采用异步调用的方式，异步调用慢慢发扬光大，出现了很多种消息队列的技术。

曾经比较主流的消息队列技术如 JMS，现在比较常用的产品有 ActiveMQ、Kafka、RabbitMQ 等，这其中 Kafka 用得比较多，下面简单介绍一下。

Kafka 来自于 LinkedIn，之后成为 Apache 项目的一部分。

Kafka 本身是一个基于分布式的系统，本身具有快速、可扩展、高性能、低耦合、高可靠性、数据不丢失等优点。

Apache Kafka 与传统消息系统相比，有以下不同：

- 基于分布式，易于向外扩展；
- 可同时为发布和订阅提供高吞吐量；
- 支持多订阅者，当失败时能自动平衡消费者；
- 将消息持久化到磁盘，因此可用于批量消费，如 ETL 及实时应用程序等。

Kafka 的整体示意图，如图 8-8 所示。

Kafka 的一些术语解释如下。

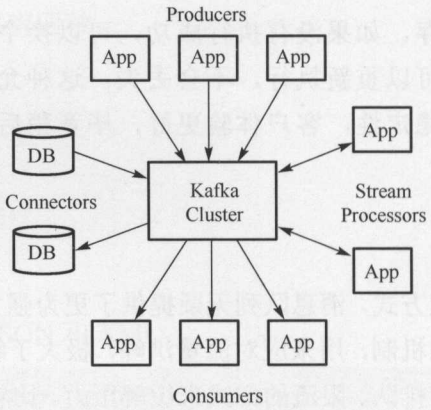


图 8-8　Kafka 示意图

Broker：Kafka 集群包含一个或多个服务器，这些服务器被称为 broker。

Topic：Kafka 集群上有很多个 Topic，每个 Topic 代表一类消息。所有发布到集群上的消息都会进入到属于自己的那个 Topic 之中，等待被处理。

Partition：在物理存储上，每个 Topic 包含一个或多个 Partition，创建 topic 时可指定 Partition 的数量。每个 Partition 对应于一个文件夹，该文件夹下存储该 Partition 的数据和索引文件，如图 8-9 所示。

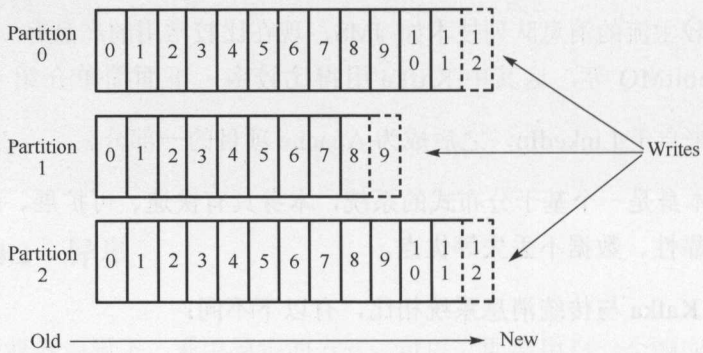


图 8-9　Partition 图

Producer：负责发布消息到 Kafka broker。

Consumer：消息消费者，负责从 Kafka broker 读取消息进行处理。

Consumer Group：每个消费者都属于一个特定的 Consumer Group，没有设定就是默认组。如图 8-10 所示。

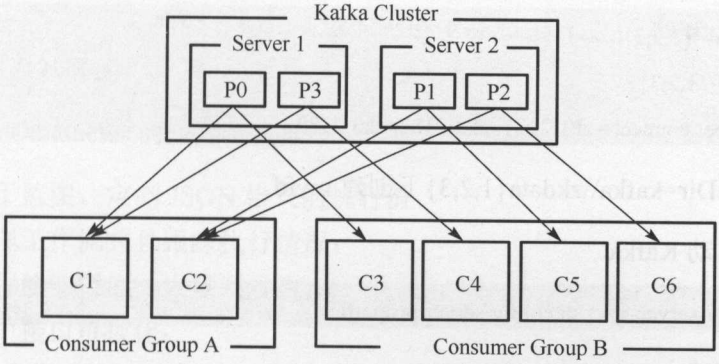


图 8-10　Consumer Group 图

下面介绍一下 Kafka 的使用。

Kafka 与 Zookeeper 结合起来使用，使用步骤如下。

材料准备：

```
Kafka 2.11-0.9.0
Zookeeper:3.4.9
```

（1）配置 zookeeper。

修改 hosts 文件：

```
192.168.1.101    zk1
192.168.1.102    zk2
192.168.1.103    zk3
```

修改 zookeeper.properties 配置：

```
server.1=zk1:2888:3888
server.2=zk2:2888:3888
server.3=zk3:2888:3888
```

（2）启动 zookeeper。

```
bin/zookeeper-server-start.sh conf/zookeeper.properties
```

（3）配置 Kafka 集群。

```
Config/server.properties
```

```
Broker.id={1,2,3}
Port=909{1,2,3}
Zookeeper.connect= zk1:2181, zk2:2182, zk3:2183
```

在 dataDir=kafka\\zkdata{1,2,3}下创建 myid。

（4）启动 Kafka。

```
bin/kafka-server-start.sh config/server.properties
```

（5）创建 test topic。

```
bin/kafka-topics.sh --create --zookeeper zk1:2181 --replication-factor 1 --partitions 1 --topic test
```

（6）查看 topic。

```
bin/kafka-topics.sh --describe --zookeeper zk1:2181
```

（7）启动生产者和消费者。

```
bin/kafka-console-producer.sh --broker-list zk1:9091, zk2:9092, zk3:9093 --topic test
bin/kafka-console-consumer.sh --zookeeper zk1:2181 --topic test --from-beginning
```

8.5 服务编排

微服务里面另一个高频词是服务编排。服务编排就相当于指挥。指挥的重要性不言而喻，无论是一个乐团的演出，或者一支军队参加的战役，指挥都是最重要的角色。服务就像乐团里的乐手，各自演奏自己的乐器，他们之间彼此独立，谁先演奏谁后演奏，什么时候一起演奏都是由乐团指挥决定的。

一个微服务架构当中服务的数量可能成百上千，其中某些单独的服务不足以实现一个完整的业务功能，必须要通过调用其他服务来完成，彼此间的调用关系由指挥来统领。就像活字印刷要有人把所有胶泥制字按照一定顺序组合好。

在《西游记》当中，九九八十一难串起了整个取经的业务流程，每一难都是一个业务环节，每一个业务环节当中又涉及多个服务的调用，如孙悟空要请哪个神仙来帮忙打妖怪，这些都需要服务编排。

下面介绍一个专业的服务编排引擎 Netflix Conductor，是由 Netflix 公司开发并贡献给开源社区的。

Netflix Conductor 的一些特点如下：

- 基于蓝图，定制 JSON 格式的执行流；
- 管理工作流并且跟踪执行情况；
- 流程能够暂停、恢复及重启；
- 用户接口可视化；
- 可强制同步处理所有任务；
- 支持百万级别的工作流；
- 基于 HTTP 或其他传输协议，如 gRPC。

Netflix Conductor 可以跨服务实现编排，同时提供对其交互的控制和可见性。具有编排跨微服务的能力，也可以帮助我们利用现有服务来构建新流或更新现有流。它的体系结构图如图 8-11(a)所示。

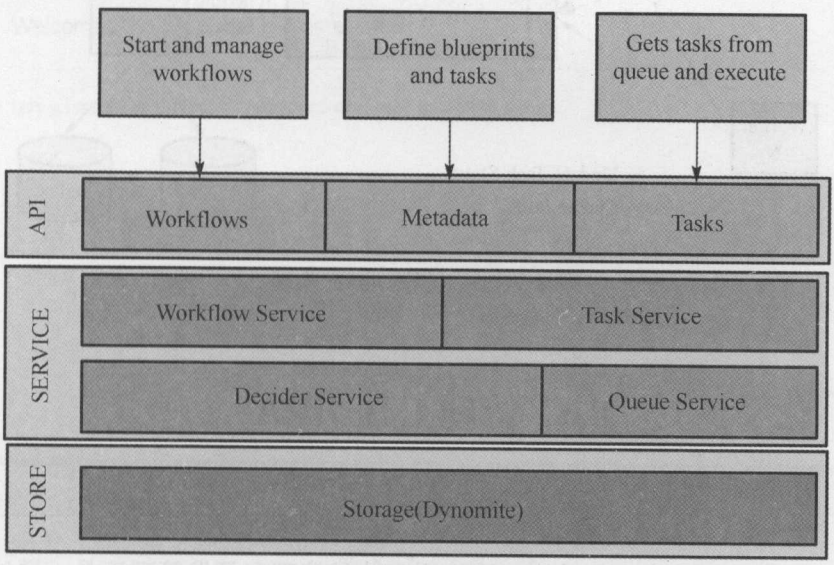


图 8-11(a)　conductor 体系结构图

Netflix Conductor 的一些概念如下。

Workflow：开始管理。

Metadata：定义蓝图和任务。

Tasks：从队列获取任务并执行。

Task：定义多个任务。

Workflow：定义任务之间的关系，执行顺序。

task worker 的实现方式如图 8-11(b)所示，有两种实现方式：一种是被编排引擎调用，通过 Rest 方式发送 POST 请求；另一种是通过定时轮询的方式到队列里获取任务。

使用 Dynomite 作为存储引擎，能够支持多种不同的存储系统，包括传统的 RDBMS 或 Apache Cassandra 这种 no-sql 存储。

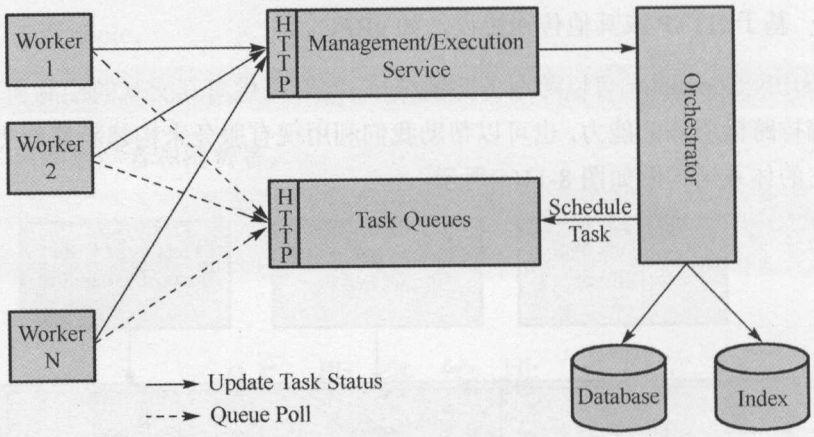


图 8-11(b)　任务 worker 实现

8.6　管理工具 JIRA

JIRA 被业界公认为是最好的敏捷团队项目管理和开发管理工具，通过它可以管理需求、问题等。

整个团队通过看板功能可以很直观地了解目前的工作进展及下一步的工作计划，使整个团队能有一个最大的输出。通过敏捷看板，可以让团队专注于提供迭代和增量价值。

JIRA 包含了很多内容，除了需求管理和缺陷管理，其他还有如下内容：

- 各种业务管理和软件管理；
- 可订制化工作流程；
- 作为持续集成的一部分，与其他开发工具集成；
- 移动端的支持；
- 有丰富的插件可供使用。

在微服务架构开发过程中需要借助 JIRA 来管理需求、问题和进度等。

下面介绍一下 JIRA 云，使用起来更加方便，无须安装直接使用。界面也较之前大为不同，如图 8-12 所示。选择一个 avata 之后就可以创建工程了，如图 8-13 所示。更详细的工程类型如图 8-14 所示。可以看出分为软件和商业两大部分，其中软件部分包括 scrum 开发、看板开发和基本软件开发。

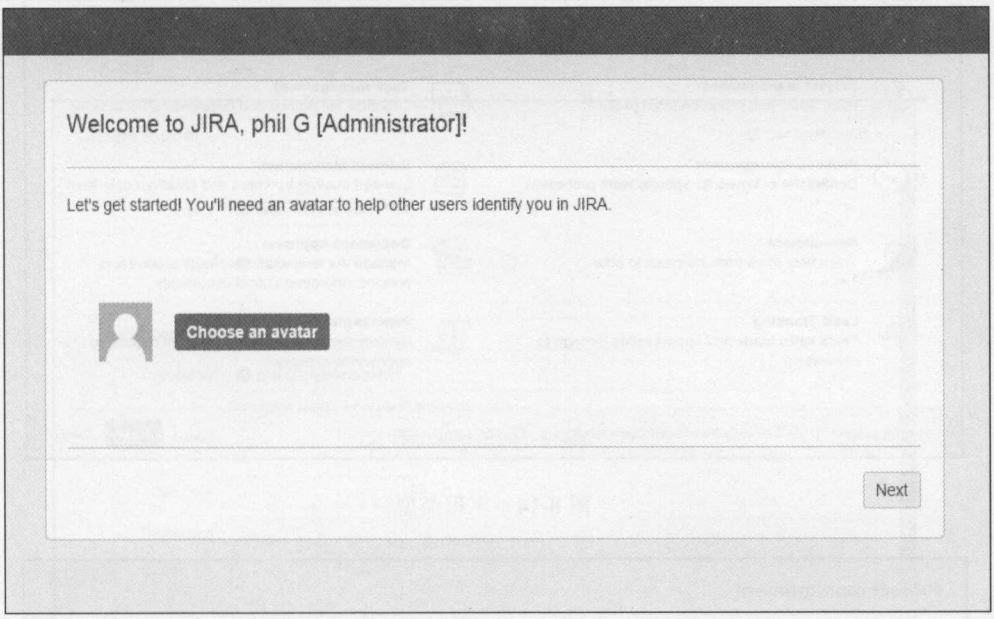


图 8-12　选择化身

商业包括的内容更多，包括项目管理、过程管理、招聘管理、机会管理等，非常丰富。

创建一个项目管理，如图 8-15 所示。工作流分为将要做、正在做和已经做完。



图 8-13　创建工程

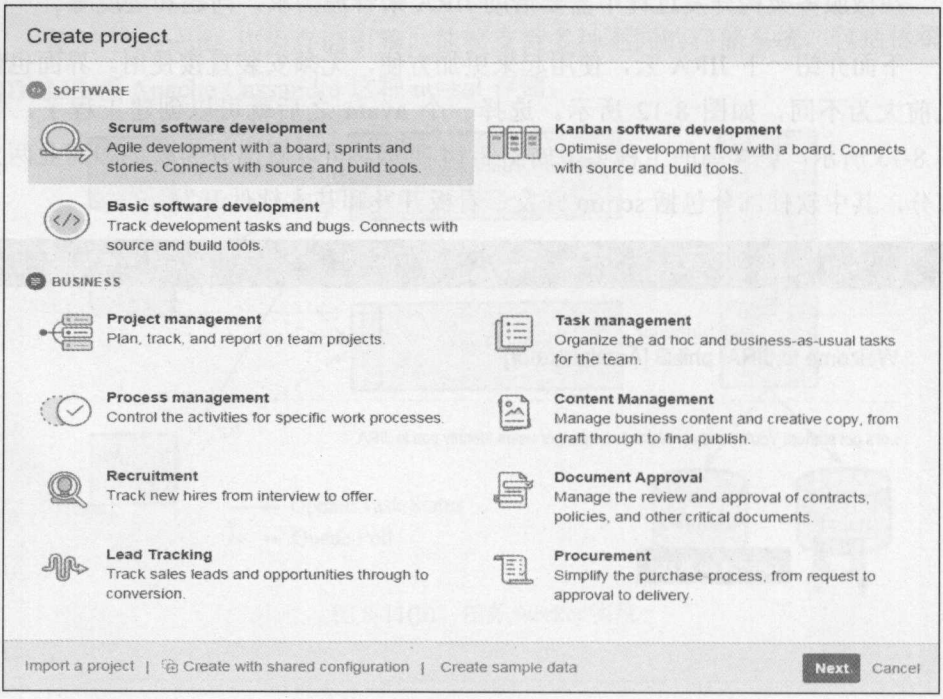


图 8-14　工程类型

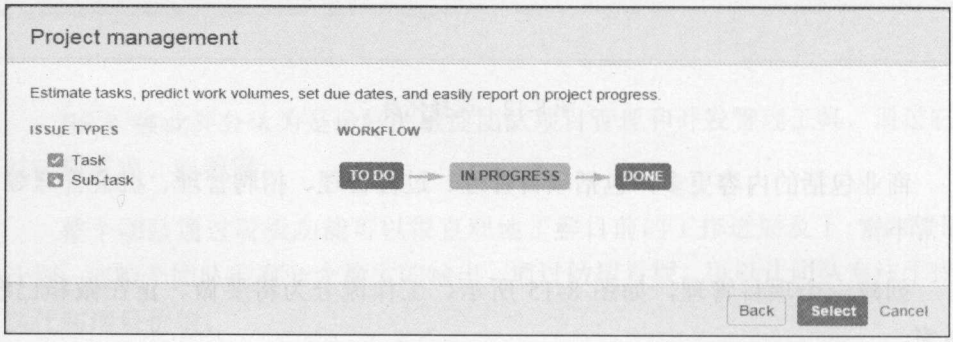


图 8-15　项目管理

点击 select 按钮后，显示工作空间的界面，如图 8-16 所示。

再点击 Greate 按钮，就可以创建 Issue 了，如图 8-17 所示。

以上只是简单介绍一下 JIRA 的使用，其功能很强大，读者可以自行去开发。

图 8-16　工作区

图 8-17　创建问题



测试相关

微服务由于自身分布式的特点，服务数量很多，互相之间的调用关系会更复杂，接口数量也会大幅度增加，导致测试的复杂度增加。

测试的过程是由小及大，从最小粒度单元测试到最大粒度单元的全流程测试。

首先要对每一个服务中的方法进行单元测试，这是测试的最小粒度，保证每一个方法都是正常的。

当单元测试完成之后进行组件测试，接下来是接口测试，因为每一个服务对外提供接口 API，每一个 API 都相当于一个商品，提供一个完整的服务，需要验证所有接口的正常使用。因为接口不只提供给前端，还要供其他服务调用，接口本身的高可用性非常重要。

本章主要讲解单元测试和接口测试及一些测试工具的使用。

9.1 单元测试

大家在软件工程上的一个共识是，如果程序的每个模块都是正确的，模块与模块的连接是正确的，那么程序本身基本上就是正确的。所以单元测试的概念就是保证构成程序的每个模块的正确性，从而保证整个程序的正确性的方法论。

在微服务架构中，服务之间存在调用关系，这时如果一个底层的 infrastructure service 存在 BUG，会导致所有调用它的 product service 都失败，这样的影响范围就会非常大，那么如何避免这种情况呢？

首先要保证每一个服务的代码质量都非常高，而途径就是通过编写单元测试来实现，通过编写正向、反向等多种测试 case 保证一个类中的每一个方法都被覆盖。

常用的单元测试框架是 Junit，为了配合单元测试还需要 Mock。

Mock 测试是单元测试的重要方法。

Mock 测试就是在测试过程中，对于某些不容易构造（如 HttpServletRequest 必须在 Servlet 容器中才能构造出来）或者不容易获取的比较复杂的对象（如 JDBC 中的 ResultSet 对象），用一个虚拟的对象（Mock 对象）来创建以便于进行测试的测试方法。

Mock 最大的功能是帮助把单元测试的耦合分解开，如果代码对另一个类或接口有依赖，它能够帮助模拟这些依赖，并帮助验证被调用的依赖的行为。

目前 Mock 测试最好的框架是 Mockito，它的 API 很少，所以学习门槛很低，可以快速上手。

Mockito 通过方法 when()、thenReturn()等可以对方法进行打桩，让后续方法按照自己的数据桩来返回，达到了隔离依赖的效果。它不需要 expectation 的概念，只需要打桩即可，一旦打桩，该方法将始终返回一个打桩的值，无论它有多少次被调用。

Mock 的其他一些特点如下：

- 可以 Mock 任何对象；
- 使用非常简单；
- 验证结果更加自由；
- 可以验证行为、顺序等；
- 可以连续调用。

下面介绍一下 Mockito 框架的使用。



9.1.1 Mockito 如何使用

第一步：在 pom.xml 增加依赖、Junit 和 mockito。

```
<plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <dependencies>
            <dependency>
                <groupId>org.apache.maven.surefire</groupId>
                <artifactId>surefire-junit47</artifactId>
                <version>${surefire.version}</version>
            </dependency>
        </dependencies>
        <configuration>
          <excludedGroups>${surefire.excludedGroups}</excludedGroups>
            <forkCount>1</forkCount>
            <reuseForks>true</reuseForks>
            <runOrder>alphabetical</runOrder>
        </configuration>
</plugin>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>${mockito.version}</version>
    <scope>test</scope>
</dependency>
```

第二步：在路径 src/test/java 下新建 package，创建测试类 xxxWebServiceTest。

整个测试类包含了以下几项。

（1）首先是注解部分增加以下几项配置：

@RunWith(SpringJUnit4ClassRunner.class);

@ContextConfiguration(classes = { TestContext.class });

@WebAppConfiguration。

（2）然后引入测试工具类。

通过@Mock 和@InjectMocks 来注入需要 Mock 的对象，然后就可以进行单元测试代码的编写和测试了。

在测试时开发人员经常需要手工造一些模拟数据，用于校验代码是否正确，对于一些数据实体比较简单的测试相对容易，一旦数据实体包含的字段特别多时工作量就会特别大，为了减少这种不必要的测试工作量可以通过 PodamFactory 来解决。

PodamFactory 是用于自动补全实体对象的类，节省了开发人员编写测试代码时手动填充数据的问题，节省了大量时间，十分便利。

（3）然后是通过注解@before 设置启动之前 setUp，将需要使用的工具类准备好，就可以开始测试了。

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = { TestContext.class })
@WebAppConfiguration
public class xxxWebServiceTest {
    @Mock
    private Logger lOG;
    @Mock
    private XxxService xxxService;
    @InjectMocks
    private XxxWebService xxxWebService;
    private MockMvc mockMvc;
    private PodamFactory podam;

    @Before
    public void setUp() throws Exception {
        MockitoAnnotations.initMocks(this);
        xxxWebService = new XxxWebService();
        podam = new PodamFactoryImpl();
        ReflectionTestUtils.setField(xxxWebService,
                "xxxService", xxxService);
        this.mockMvc = MockMvcBuilders
                .standaloneSetup(xxxWebService).build();
```

```
    }
}
```

以本书 8.3 节作为例子进行测试。

```
@RunWith(SpringJUnit4ClassRunner.class)
@ActiveProfiles(profiles = "dev")
@SpringApplicationConfiguration(classes = {
TestContext.class,
WebSecurityConfig.class,
Http401UnauthorizedEntryPoint.class
        })
@WebAppConfiguration
public class SecurityConfigurationTest {
    @Inject
    private ConfigurableWebApplicationContext context;
    private MockMvc mvc;

    @Before
    public void setup() {
        mvc = MockMvcBuilders.webAppContextSetup(context)
                .apply(springSecurity()).build();
    }
```

MockMvc 可以模拟客户端发起 WebService 请求，对 controller 中的一次调用进行模拟，perform 就是一次请求，MockMvcRequestBuilders 进行 url 的请求，

andExcept 方法为对 Controller 类、调用方法、视图和 model 的预期设置，andDo 进行这次请求的执行，最后 andReturn 返回。

MockMvc 示例如下。

（1）校验 rest 路径地址不存在。

因为这个地址是不存在的，所以和预期的返回的结果 NotFound 相同，这样测试就通过了。

```
    @Test
    public void withoutUserHasAccessToPublicEndpoint() throws Exception {
        mvc.perform(get("/public/noexist")).andExpect(status().isNotFound());
    }
```

（2）校验没有权限。

因为在权限篇，已经对/api/** 路径设置了 authenticated()，所以当请求这个地址时必须要有权限才可以，直接调用肯定是没有权限无法通过的，所以下面的测试预期结果就是 Unauthorized，这样测试就通过了。

```
@Test
public void withoutUserHasNoAccessToSecuredEndpoint() throws Exception {
    mvc.perform(get("/api/rest")).andExpect(status().isUnauthorized());
}
```



9.1.2 Mock 与 InjectMocks 的区别

注解当中比较常用的分为@Mock 和@InjectMocks，都是关于 Mock 的，二者有什么区别呢？

举例来看。

此处要测试一个 service 的方法，service 会调用一个 repository，repository 类似于一个 DAO，执行数据库的操作，可以看到代码中 repository 采用的是 Mock 注解，而 service 采用了 InjectMocks 注解。

@Mock 注解开启了 mock 功能，repository 变成了一个 Mock 对象，然后对它进行 Mock，如使用 save 方法时则返回一个对象。它不会向下执行。

而 service 方法需要调用 repository，它需要向下执行，返回结果，这时就必须用@InjectMocks，否则它就直接返回 mock 对象了，不会调用 repository。

第一步：首先准备一个 service 和一个 repository。

（1）service 代码如下。

```
@Component
public class XxxServiceImpl {
    @Override
    public void createXxx(ProductOrdersDTO poDTO) {
        ProductOrders poEntity = xxxServiceImpl
                . xxxDTOToEntity (poDTO);
        if (poEntity!= null) {
            xxxRepository.save(poEntity);
        }
```

```
}
}
```

（2）repository 代码如下。

```
public interface XxxRepository extends
            PagingAndSortingRepository< ProductOrders, Long> {
  Page findAll(Pageable pageable);
 }
```

（3）然后通过测试代码进行测试，代码如下。

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = { TestContext.class })
public class XxxServiceImplTest {
      @Mock
      private XxxRepository xxxRepository;
      @InjectMocks
      private XxxServiceImpl xxxServiceImpl;
       private PodamFactory podam;
      @Before
      public void setUp() throws Exception {
            MockitoAnnotations.initMocks(this);
            podam = new PodamFactoryImpl();
            xxxServiceImpl = new XxxServiceImpl ();
            ReflectionTestUtils.setField(xxxServiceImpl,
                        " xxxRepository ", xxxRepository);
    }

      @Test
      public final void testCreateProductOrdersDTO () throws Exception {

            ProductOrders poEntity = podam
                        .manufacturePojo(ProductOrders.class);
            ProductOrdersDTO poDTO = podam
                        .manufacturePojo(ProductOrdersDTO.class);

            when(xxxServiceImpl.xxxDTOToEntity(poDTO))
                              .thenReturn(poEntity);
```

```
        when(xxxRepository.save(poEntity))
                .thenReturn(poEntity);

        this.xxxServiceImpl.createXxx(poDTO);

        verify(xxxRepository, times(1)).save(eq(poEntity));
    }
```

9.2 接口测试

微服务架构是面向服务的架构，所有应用都是以接口的形式对外提供服务，那么就会自然地形成前后端分离，以及多个系统之间相互调用的关系，前后端以及服务之间基于接口的集成和联调都是不可避免的。

在一开始双方就需要定一个契约，就像买卖双方签订的合同，双方就 API 相关的内容，包括路径、类型、参数、请求方式、返回值等达成一致。

路径就相当于网店地址，类型就是产品包装（JSON 还是 XML），参数是支付形式和金额，请求方式（Rest 还是 RPC），以及最重要的返回值（商品是不是爆款）。

如果只是一次性购买，问题倒不大，新品总会上架，就像 API 总是会变的一样。产品如果没有设计好，就像 API 没设计好，很有可能会频繁地修改。

这时如何即时通知买家，让买家知道出新款（版本）了，价格（参数）也调整了。要让买家知道最新的合同内容是最重要的，不然没有办法正常获得商品。

接口调整之后相当于重签了合同，这时如何保证所有涉及的系统步调一致、整齐划一，在持续集成、持续交付的过程中，如何保证所有系统不会像路痴队友一样掉队，都是要解决的问题。

所以接口测试又经常被称为契约测试，测试的内容就是买方通过正确的地址，提供正确的参数，获得卖方提供的正确的产品，保证合同约定的和实际交付的是一样的。

契约测试可以通过一些工具，详情请看本书 8.2 节介绍的利用 Swagger 进行接口测试。也可以通过浏览器直接发起请求，验证返回结果。

总之，进行契约测试的目的就是尽早地发现版本差异性，并做出调整，将最后集成的风险降到最低。

9.3 代码质量管理工具 Sonar

因为拆分成微服务之后，要保证各个服务的代码质量，必须要让代码的单元测试覆盖率达到一定比例，如有的公司要求 90%以上。达到这个标准才是合格的，否则算失败，那么通过什么工具来管控呢？

这个时候就需要代码质量管理工具了，而 Sonar 是目前最强大的代码质量管理工具之一。

Sonar 是一个用于代码质量管理的开放平台。通过插件机制，Sonar 可以集成不同的测试工具、代码分析工具，以及持续集成工具，如 Jenkins。



9.3.1 特点

Sonar 支持多语言，总共支持超过 20 种以上的语言分析工作，不仅包含编程语言、脚本语言，还有数据库语言。

Sonar 主要特点如下。

- 通过设置质量门槛，能很直观地看到工程的质量是否达标，相当于在整个软件的开发过程中始终有一个量尺在跟踪工程的质量。
- 可以看到总体健康程度，以及专门针对测试覆盖率和重复的指标，因为在软件工程中这两个指标是最重要的。
- 通过代码规则检测工具检测不符合标准的代码以及潜在的缺陷和 BUG。
- 检测注释不足或过多的情况。
- 统计代码行数。
- 整合 DevOps。
- 集中管理质量。

利用这些特点更利于编写干净整洁的代码，通过没有单元测试覆盖的代码及复杂度高的代码这两种指标来判断可能存在 BUG 的情况。

默认的规则设置可以满足开发者对于质量的要求，同时又可以自定义规则来更好地满足自身的需要。

Sonar 可以遍历所有方法的执行路径，当调用分支比较多时，组合数量大，手工测试工作量大时，这时 Sonar 可以测试所有组合的结果。

同时 Sonar 是 DevOps 生态的一部分，可以通过 Rest API 整合到持续集成当中。

最后不同部门和整个机构都可以在同一个地方看到项目的质量管理规则，通过定义好规则集，每个项目组可以根据自己的需要选择规则集。

实现跨项目服务，一个用户可以在系统中被分配多个工程，通过这一功能可以对所有有关项目进行查看，而不仅限于一个项目。建立统一的质量门槛，保证不同类型的工程质量始终如一。

它提供基于风险的视图，根据风险点来识别出潜在风险，在风险爆发之前进行处理。



9.3.2 使用

Sonar 官网地址：https://www.sonarqube.org/。打开界面如图 9-1 所示。

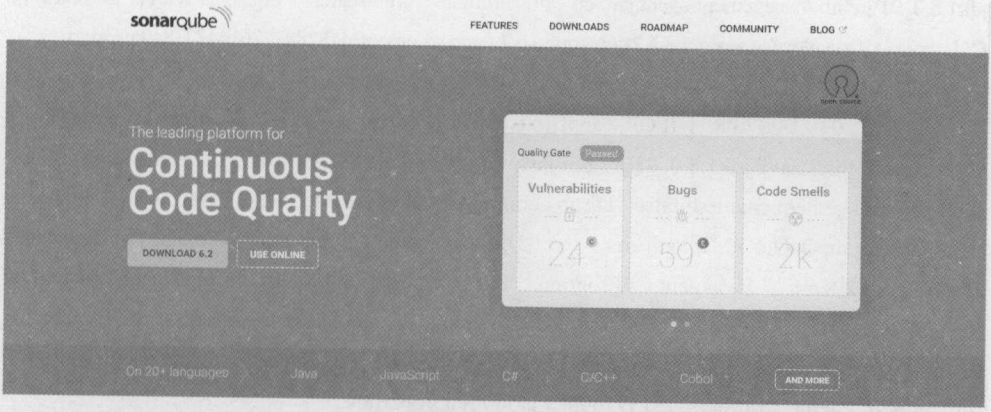


图 9-1 官网图

第一步：下载最新版的 SonarQube，当前最新版是 6.2。

（1）在 conf 目录下的 sonar.properties 里修改成本地数据库参数。

sonar.properties 修改的地方如下。

```
sonar.jdbc.url=jdbc:mysql://localhost:3306/sonar?useUnicode=true&characterEncoding=utf8
&rewriteBatchedStatements=true&useConfigs=maxPerformance
sonar.jdbc.username=xxx
sonar.jdbc.password=xxx
```

（2）然后启动 Sonar，当显示三个 is up 之后程序就已经成功启动了，三个 up 分别是 Compute Engine、Elasticsearch 和 Web。

启动日志如下。

```
--> Wrapper Started as Console
Launching a JVM...
Wrapper (Version 3.2.3) http://wrapper.tanukisoftware.org
  Copyright 1999-2006 Tanuki Software, Inc.  All Rights Reserved.
2017.02.03 14:36:38 INFO  app[][o.s.a.AppFileSystem] Cleaning or creating temp directory
C:\Users\xxx\Downloads\sonarqube-6.2\sonarqube-6.2\temp
2017.02.03 14:36:38 INFO  app[][o.s.p.m.JavaProcessLauncher] Launch process[es]:
C:\Program Files\Java\jdk1.8.0_91\jre\bin\java -Djava.awt.headless=true -Xmx1G -Xms256m
-Xss256k -Djna.nosys=true -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:CMSInitiating-
OccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly -XX:+HeapDumpOnOutOfMemoryError
-Djava.io.tmpdir=C:\Users\xxx\Downloads\sonarqube-6.2\sonarqube-6.2\temp -javaagent: C:\Program Files\Java\
jdk1.8.0_91\jre\lib\management-agent.jar -cp ./lib/common/*; ./lib/search/* org.sonar.search. SearchServer
C:\Users\xxx\Downloads\sonarqube-6.2\sonarqube-6.2\temp\sq-process4782882276144777849 properties
2017.02.03 14:37:08 INFO  app[][o.s.p.m.Monitor] Process[es] is up
2017.02.03 14:37:08 INFO  app[][o.s.p.m.JavaProcessLauncher] Launch process[web]:
C:\Program Files\Java\jdk1.8.0_91\jre\bin\java -Djava.awt.headless=true -Dfile.encoding=UTF-8
-Djruby.management.enabled=false -Djruby.compile.invokedynamic=false -Xmx512m -Xms128m
-XX:+HeapDumpOnOutOfMemoryError -Djava.io.tmpdir=C:\Users\xxx\Downloads\sonarqube-6.2\
sonarqube-6.2\temp -javaagent:C:\Program Files\Java\jdk1.8.0_91\jre\lib\management-agent.jar
-cp ./lib/common/*;./lib/server/*;C:\Users\xxx\Downloads\sonarqube-6.2\sonarqube-6.2\lib\jdbc\my
sql\mysql-connector-java-5.1.39.jar org.sonar.server.app.WebServer C:\Users\xxx\Downloads\sonarqube-6.2\
sonarqube-6.2\temp\sq-process1513776908754549290properties
2017.02.03 14:39:28 INFO  app[][o.s.p.m.Monitor] Process[web] is up
2017.02.03 14:39:29 INFO  app[][o.s.p.m.JavaProcessLauncher] Launch process[ce]:
```

```
C:\Program Files\Java\jdk1.8.0_91\jre\bin\java -Djava.awt.headless=true -Dfile.encoding=UTF-8
-Xmx512m -Xms128m -XX:+HeapDumpOnOutOfMemoryError -Djava.io.tmpdir=C:\Users\xxx10\
Downloads\sonarqube-6.2\sonarqube-6.2\temp -javaagent:C:\Program Files\Java\jdk1.8.0_91\jre\lib\
management-agent.jar
-cp ./lib/common/*;./lib/server/*;./lib/ce/*;C:\Users\xxx\Downloads\sonarqube-6.2\sonarqube-6.2\li
b\jdbc\mysql\mysql-connector-java-5.1.39.jar org.sonar.ce.app.CeServer C:\Users\xxx\Downloads\
sonarqube-6.2\ sonarqube-6.2\temp\sq-process8746534000997736345properties
    2017.02.03 14:39:46 INFO  app[][o.s.p.m.Monitor] Process[ce] is up
    2017.02.03 14:39:46 INFO  app[][o.s.application.App] SonarQube is up
```

（3）可以通过浏览器访问如下地址：localhost：9000，如图 9-2 所示。

点击 login 按钮，如图 9-3 所示。输入 admin/admin。

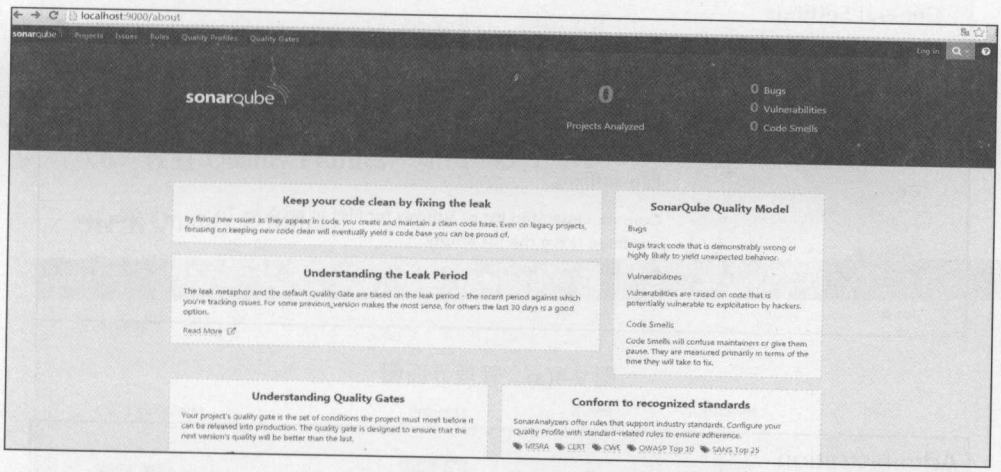


图 9-2　首页图

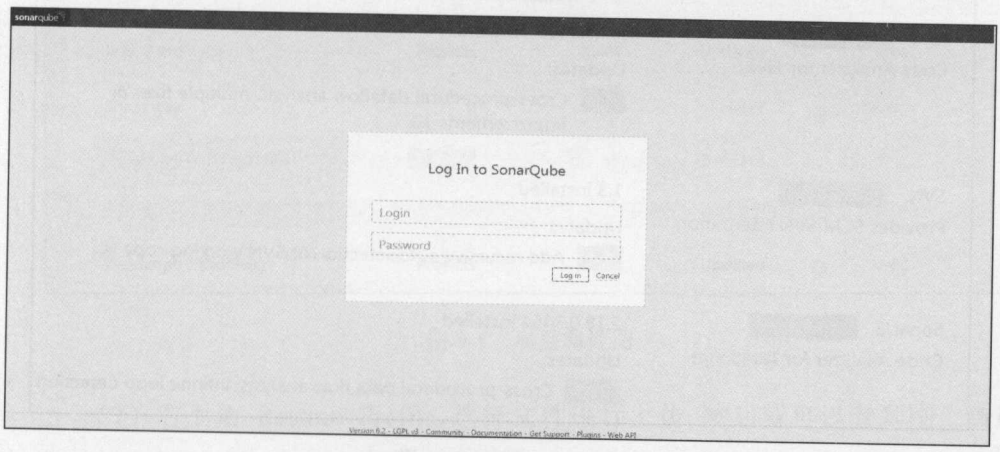


图 9-3　登陆图

（4）在管理控制台点击 System 按钮进行插件更新，更新几个项目，如图 9-4(a)、图 9-4(b)、图 9-4(c)所示。

```
sonar-java-plugin
sonar-findbugs-plugin
sonar-cobertura-plugin
sonar-checkstyle-plugin
sonar-pmd-plugin
```

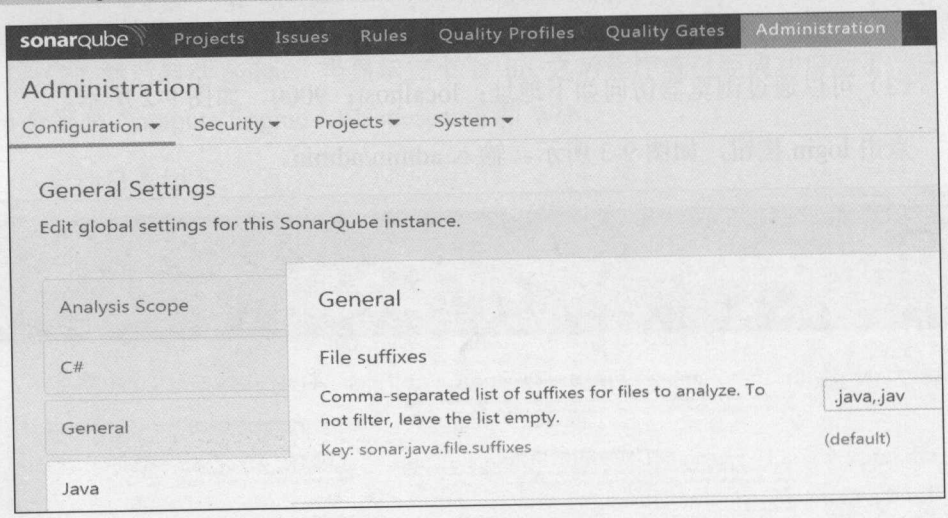


图 9-4(a)　管理界面图

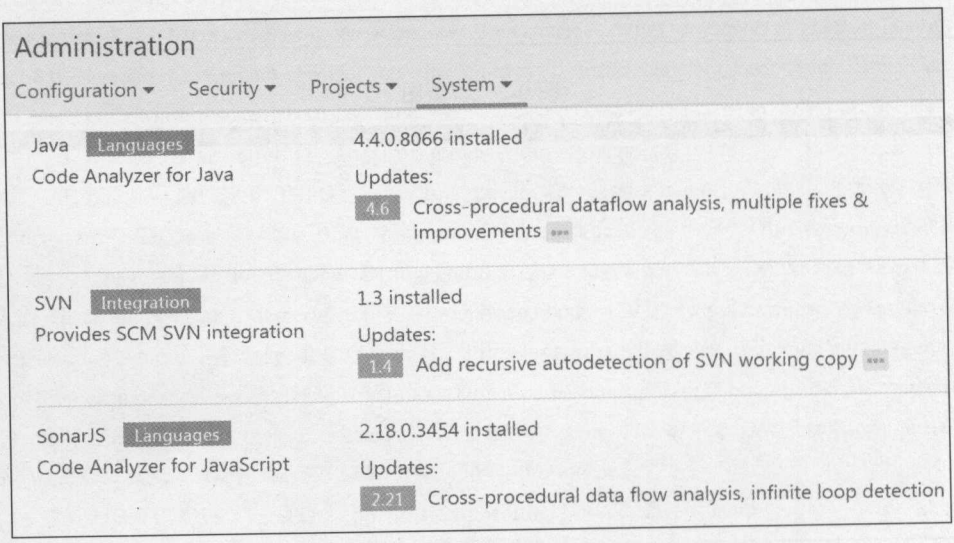


图 9-4(b)　插件列表图

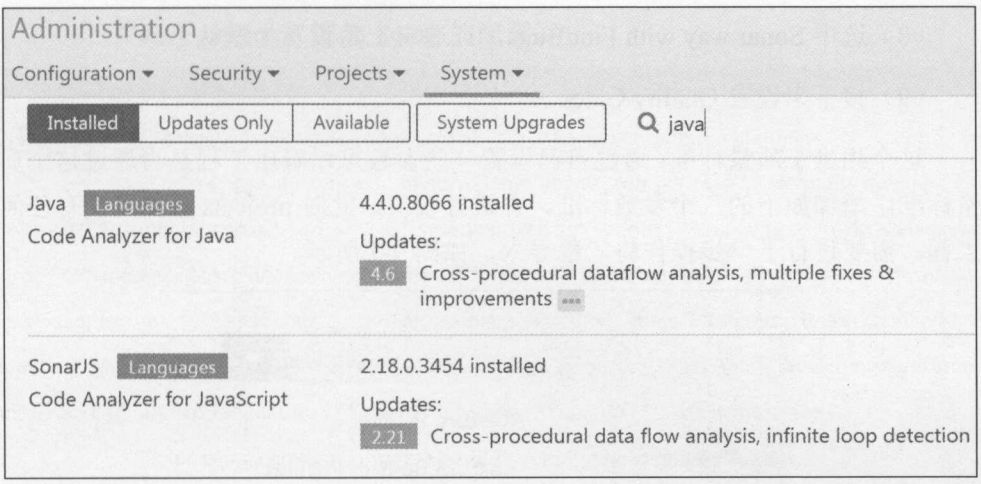


图 9-4(c)　查询结果图

（5）更新插件之后需要重启 Sonar，使其生效。

（6）设置 Quality Profiles，如图 9-5 所示。

选择 Quality Profiles。



图 9-5　质量属性图

（7）右上角 Restore Profile，选择文件进行上传，然后就可以看到刚上传的 Sonar way with FindBugs，如图 9-6 所示。

（8）选中 Sonar way with FindBugs，点选右上角设置为默认。

（9）接下来设置 Quality Gates。

这个相当于质量标准，通过自己设置一些参数可以看出工程是否通过这个质量标准。增加如下的三个参数标准，并进行设值，此时 projects 里面还没有任何工程。需要进行下一步操作将工程导入，如图 9-7 所示。

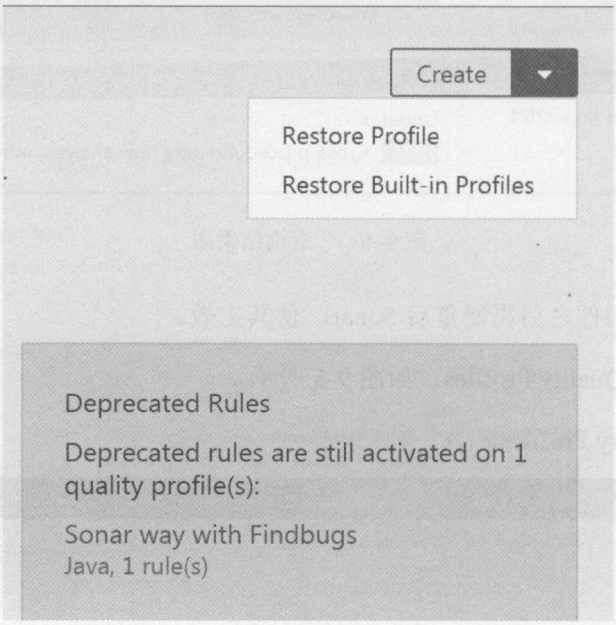


图 9-6　重置属性图

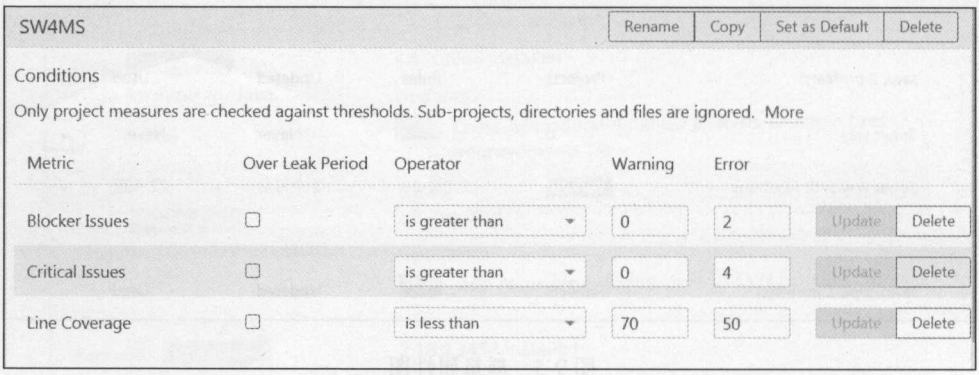


图 9-7　质量门槛图

第二步：在工程中增加 Sonar 的配置，让工程与 Sonar 建立联系。

（1）首先在 pom.xml 增加 Sonar 的配置。

```
<!-- Sonar Configuration -->
        <sonar-maven-plugin.version>2.7</sonar-maven-plugin.version>
         <sonar.exclusions>src/main/webapp/bower_components/**,src/main/webapp/
swagger/**,src/main/webapp/styles/*.css*,src/main/webapp/scripts/**/*-spec.js,src/main/webapp/sc
ripts/**/*-spec-e2e.js,src/main/webapp/scripts/**/_module.js,src/main/webapp/scripts/lib/**,src/ma
in/webapp/scripts/**/*-spec-data.js,src/main/webapp/dist/**,src/main/java/com/jzb/userservice/serv
ice/dto/**,src/main/java/com/jzb/userservice/backend/odata/proxy/**,src/main/java/com/jzb/userser
vice/backend/odata/proxy/entities/**,src/main/webapp/swagger/**</sonar.exclusions>
        <sonar.java.coveragePlugin>jacoco</sonar.java.coveragePlugin>
        <sonar.java.source>1.8</sonar.java.source>
        <sonar.junit.reportsPath>target/surefire-reports</sonar.junit.reportsPath>
        <sonar.sources>src/main</sonar.sources>
       <dependency>
            <groupId>org.sonarsource.scanner.maven</groupId>
            <artifactId>sonar-maven-plugin</artifactId>
            <version>3.0.1</version>
        </dependency>
              <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>sonar-maven-plugin</artifactId>
            <version>${sonar-maven-plugin.version}</version>
        </plugin>
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>${maven-jacoco-plugin.version}</version>
            <executions>
                <execution>
                    <id>prepare-unit-test-agent</id>
                    <goals>
                        <goal>prepare-agent</goal>
```

```
                </goals>
            </execution>
            <execution>
                <id>generate-unit-test-report</id>
                <goals>
                    <goal>report</goal>
                </goals>
            </execution>
            <execution>
                <id>prepare-integration-test-agent</id>
                <goals>
                    <goal>prepare-agent-integration</goal>
                </goals>
            </execution>
            <execution>
                <id>generate-integration-test-report</id>
                <goals>
                    <goal>report-integration</goal>
                </goals>
            </execution>
        </executions>
        <configuration>
            <excludes>
                <exclude>com/jzb/userservice/*.class</exclude>
                <exclude>com/jzb/userservice/config/**/*.class</exclude>
                <exclude>com/jzb/userservice/aop/**/*.class</exclude>
                <exclude>com/jzb/userservice/async/**/*.class</exclude>
                <exclude>com/jzb/userservice/backend/dao/**/*.class</exclude>
                <exclude>com/jzb/userservice/exception/**/*.class</exclude>
                <exclude>com/jzb/userservice/web/filter/**/*.class</exclude>
            </excludes>
        </configuration>
    </plugin>
```

```
<profile>
        <id>sonar</id>
        <activation>
            <activeByDefault>true</activeByDefault>
        </activation>
        <properties>
            <!-- SERVER ON A REMOTE HOST -->
            <sonar.host.url>http://localhost:9000</sonar.host.url>
        </properties>
    </profile>
```

（2）配置好这些之后，就可以通过命令 mvn sonar:sonar 来启动工程，这样工程就与 Sonar 进行了关联，构建成功后就可以在 Sonar 里面看到这个工程的测试覆盖率等信息。

启动日志如下。

```
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building jzb-userservice 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- sonar-maven-plugin:2.7:sonar (default-cli) @ jzb-userservice ---
[INFO] ------------  Scan
jzb-userservice
[INFO] Load server rules
[INFO] Load server rules (done) | time=98ms
[INFO] Initializer GenericCoverageSensor
[INFO] Initializer GenericCoverageSensor (done) | time=1ms
[INFO] Base dir: C:\Users\xxx\workspace\userservice
[INFO] Working dir: C:\Users\xxx\workspace\userservice\target\sonar
[INFO] Source paths: src/main
[INFO] Test paths: src/test/java
[INFO] Source encoding: UTF-8, default locale: zh_CN
[INFO] Index files
```

```
[INFO] Excluded sources:
[INFO]    src/main/webapp/bower_components/**
[INFO]    src/main/webapp/swagger/**
[INFO]    src/main/webapp/styles/*.css*
[INFO]    src/main/webapp/scripts/**/*-spec.js
[INFO]    src/main/webapp/scripts/**/*-spec-e2e.js
[INFO]    src/main/webapp/scripts/**/_module.js
[INFO]    src/main/webapp/scripts/lib/**
[INFO]    src/main/webapp/scripts/**/*-spec-data.js
[INFO]    src/main/webapp/dist/**
[INFO]    src/main/java/com/xxx/userservice/service/dto/**
[INFO]    src/main/java/com/xxx/userservice/backend/odata/proxy/**
[INFO]    src/main/java/com/xxx/userservice/backend/odata/proxy/entities/**
[INFO]    src/main/webapp/swagger/**
[INFO] Sensor JaCoCoSensor (done) | time=750ms
[INFO] Sensor XmlFileSensor
[INFO] Sensor XmlFileSensor (done) | time=2ms
[INFO] Sensor Zero Coverage Sensor
[INFO] Sensor Zero Coverage Sensor (done) | time=91ms
[INFO] Sensor Code Colorizer Sensor
[INFO] Sensor Code Colorizer Sensor (done) | time=18ms
[INFO] Sensor CPD Block Indexer
[INFO] JavaCpdBlockIndexer is used for java
[INFO] Sensor CPD Block Indexer (done) | time=290ms
[INFO] Calculating CPD for 55 files
[INFO] CPD calculation finished
[INFO] Analysis report generated in 1467ms, dir size=889 KB
[INFO] Analysis reports compressed in 5256ms, zip size=384 KB
[INFO] Analysis report uploaded in 1300ms
[INFO] ANALYSIS SUCCESSFUL, you can browse http://localhost:9000/dashboard/index/
com.jzb.userservice:userservice
[INFO] Note that you will be able to access the updated dashboard once the server has
processed the submitted analysis report
```

```
[INFO] More about the report processing at http://localhost:9000/api/ce/task?id=AVoC7WpoLkPnzJgganDj
[INFO] Task total time: 52.430 s
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 01:26 min
[INFO] Finished at: 2017-02-03T15:43:19+08:00
[INFO] Final Memory: 48M/699M
[INFO] ------------------------------------------------------------------------
```

再刷新页面，就可以看到工程的结果，如图 9-8 所示。上面的质量门槛显示 Passed 状态，说明已经通过了，下面有几个细项：BUG 数、代码异味、单元测试数量及覆盖率，还有重复代码。第一项都可以点击查看详情，使用起来非常方便，如图 9-9 所示。

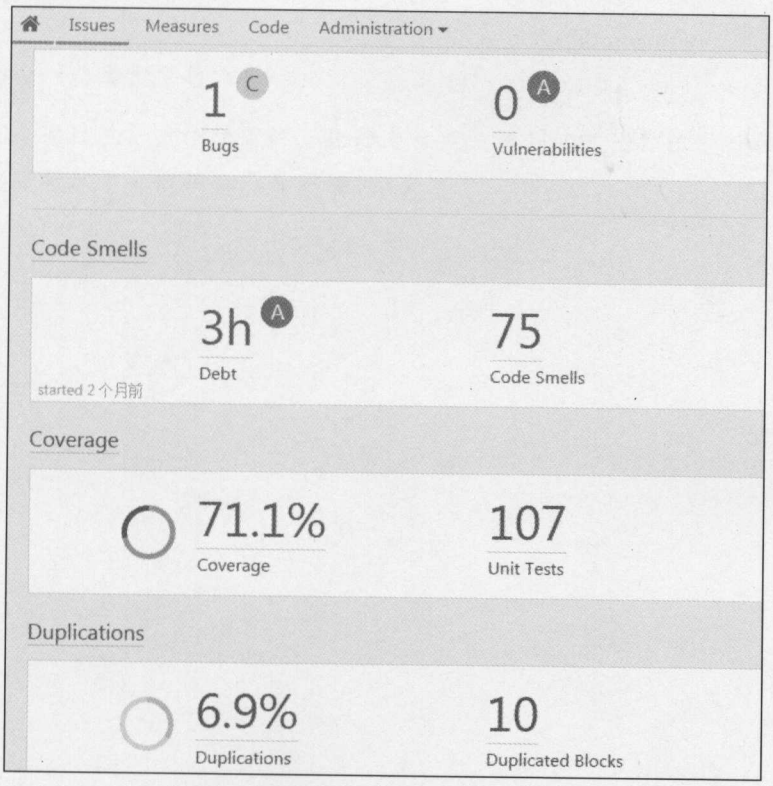


图 9-8　质量报告图

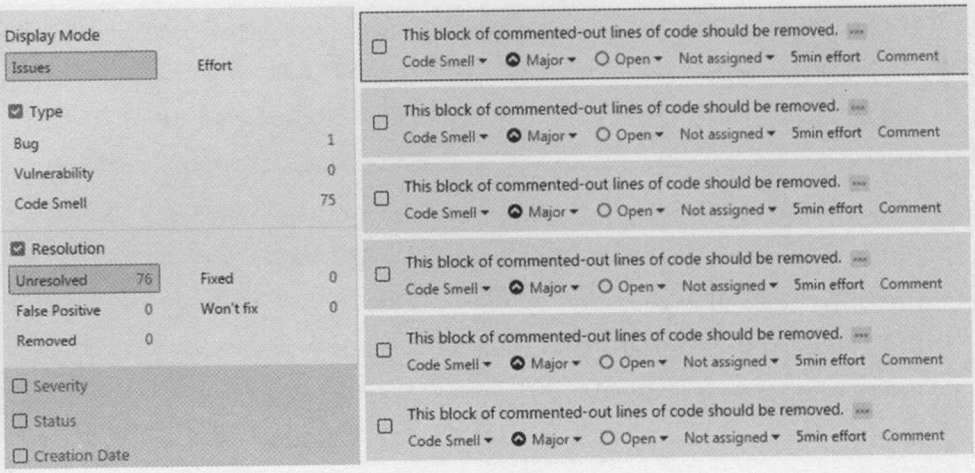

Display Mode
Issues
Effort
Type
Bug 1
Vulnerability 0
Code Smell 75
Resolution
Unresolved 76
Fixed 0
False Positive 0
Won't fix 0
Removed 0
Severity
Status
Creation Date
This block of commented-out lines of code should be removed.
Code Smell Major Open Not assigned 5min effort Comment
This block of commented-out lines of code should be removed.
Code Smell Major Open Not assigned 5min effort Comment
This block of commented-out lines of code should be removed.
Code Smell Major Open Not assigned 5min effort Comment
This block of commented-out lines of code should be removed.
Code Smell Major Open Not assigned 5min effort Comment
This block of commented-out lines of code should be removed.
Code Smell Major Open Not assigned 5min effort Comment
This block of commented-out lines of code should be removed.
Code Smell Major Open Not assigned 5min effort Comment

图 9-9　问题列表图

第三部分

项目实战篇

知行合一，止于至善。

本部分通过三个案例的讲解，让读者将前文讲到的理论知识进行融会贯通，了解在实际项目中如何拆分、如何使用。



三个典型系统案例

下面介绍三个案例如何服务化。

10.1 业务运营支撑系统

大型核心业务系统。



10.1.1 背景

系统采用 EJB 框架，已经运行十年多时间。

从最初几个月上一个业务发展到现在一个月上几十个业务。

随着业务量暴涨，人员也从十几个人的团队发展到百人的团队。

数据量呈几何级增长，拥有海量数据，整个系统像一艘巨轮。



10.1.2 现状

当前的主要问题如下。

- 架构老旧

EJB 太老太破旧，无法华丽转身。

- 系统庞大

系统没有引擎，全凭人堆，百人的团队成员成了迁夫，伴着巨轮的踯躅前行，是所有人的辛苦付出。

➢ 业务复杂

承载的业务逻辑复杂，业务之间耦合严重，代码补丁多，可读性差，新人成长慢。

➢ 无法快速交付

用户体验就是一个字“慢”，功能开发、测试和上线慢，同时上线涉及的功能模块太多，导致上线过程十分复杂。

➢ 交付质量低

运维问题多，运维问题占用了开发人员大量的时间，导致开发时间减少，只能加班开发，造成恶性循环。

➢ 海量业务数据

已经采用分库分表的方式，未来的数据量会更大。



10.1.3 原因调查

水是有源的，树是有根的，系统现在这样也是有原因的。

最主要的原因在架构上，它太老了，无法满足现在的要求。

因为 EJB 专门为企业级大型业务系统所定制，是一个大而全的架构，虽然优点很多，但是由于系统太大，就像粗犷的汉子，与精致绝缘。

遥想当年，EJB 绝对是最好的选择，大型企业级应用的首选。EJB 呼风唤雨的时候 Spring 还只是一个黄口小儿，EJB 曾拥有众多成熟案例，而如今曾经大而全的架构所带来的优势反而成了劣势，变成了系统本身的瓶颈。这也是 EJB 最大的缺点：不够灵活、太笨重。

而曾经轻量级的 Spring 虽然也越来越大，但是好在可以自由组合，随意增减，按需所取，并没有走上 EJB 的老路。面对日益膨胀的系统，EJB 这个架构已经落伍了，跟不上时代的发展，注定要被后浪拍在沙滩上。

另一个原因是业务复杂，如果业务简单，EJB 也可以应付得来，由于业务复杂，并且不断变化，架构无法应对。随着发展，新需求层出不穷，很多新业务的处理规则和以往业务完全不一样，这时候硬把这些业务塞到原有系统当中本身就不合理，就像削足适履一样，最终业务和功能之间的互相妥协造成了原系统消化不良，破坏了本身架构的平衡。

团队的船长拥有丰富的经验，在船长的英明领导之下，以及无所不知的“业务大牛”和勤勤恳恳的“技术大牛”的配合下，系统之船才始终在正确的航向上行驶，没有偏离，没有发生重大事故，虽然小问题比较多，但都在可控范围之内。

也正是因为大家的共同努力、兢兢业业才让所做的工作不断地得到客户方的认可。如果换作其他人，管理这么大的项目，这么多人的团队，问题远不止于此，后果无法想象，项目可能早就进行不下去了。此 EJB 之罪，非战之罪，这个锅必须要由 EJB 来背。

有人会问，为什么不早点换架构？

事实上团队成员也一直在寻找替代方案，但无奈在部署方面 EJB 还是有其优势的，如在集群部署方面。由于当时一直没有成熟的架构可以替代，同时新需求的开发量一直有增无减导致重构之路一直没有真正成行。

而现在微服务的出现彻底改变了情况 ，同时系统也到了不得不重构的时候，客观条件具备了，这时终于可以踏上重构之路了。



10.1.4 解决方案

首先架构层面，采用微服务架构，技术实现采用 Spring Boot+Spring Cloud 替换原有 EJB。

其他改造点如下。

- 将原来接口之间调用的 Soap 方式替换成更轻量的 Rest 方式。
- 将原有的消息队列实现方式 Jms 替换成对分布式支持更好的 Kafka。
- 将原来的数据库 Oracle 替换成 HANA 内存数据库，读写更快，突破传统数据库都会遇到的 IO 瓶颈。

因为系统十分庞大，业务十分复杂，光是介绍这个系统都可以写一本书了，所以这里只选择其中的订单业务功能来讲解，以达到举一反三的效果。下面来看具体的步骤。

第一步：梳理所有的业务功能环节。

通过将数据模型和业务模型结合起来使用，对业务进行梳理，在业务拆分方面，首先按照业务环节进行粗粒度的拆分，划分成几个大部分。

第二步：选取其中订单业务功能进行从上到下的拆分，然后在金字塔结构图上进行展示，金字塔结构图如图 10-1 所示。

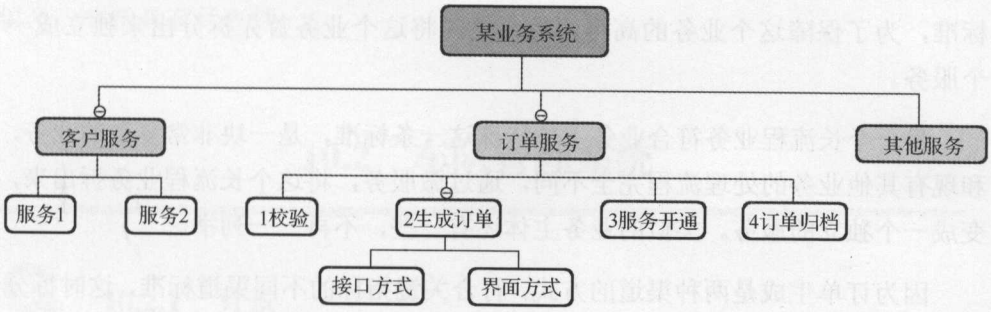


图 10-1　某业务系统图

创建订单的步骤是先校验，再生成订单，接着送开通，最后归档，所以在金字塔结构图中依次将其罗列。

创建订单的第一步是校验，针对于不同的业务主体有很多校验的逻辑，只有校验通过之后，才可以向后流转，否则直接返回失败。

接下来在生成订单时，原系统可以通过两种方式进行处理，这时将生成订单服务向下细分：一种是通过界面，一种是通过接口。

创建订单之后会根据不同业务类型送开通，并进行归档。

至此，我们便得到一个业务功能的金字塔结构图，可以在图上看到订单服务的层级情况，既可以选择使用最底层的粒度，也可以选择粗的粒度，选择哪一层级要结合考虑项目的具体情况。

第三步：梳理所有业务主体，进行归类，画出业务泳道图。因为系统的业务主体非常多，在处理逻辑上有差异，所以将业务主体分成几类，如普通业务、短流程业务、长流程业务、特殊业务等。

第四步：将金字塔结构图结合业务泳道图通过关键指标来识别关键业务功能。

首先将所有通用的业务功能都拆分出来，形成公共服务。

系统里很多环节都有校验的逻辑存在，它就属于一个公共服务，被所有业务调用，这里将校验逻辑单独提取成一个服务。由于业务规则太多，导致原系统配置太多，每次新业务都要有大量的配置工作，规则符合公共服务的标准，抽象成规则服务来实现。

拆分完公共服务之后就该拆业务主体了。

有一个业务主体，得到了各方面的重点关注，符合前文介绍的重点业务的标准，为了保障这个业务的高可用性，这时将这个业务首先拆分出来独立成一个服务。

另一个长流程业务符合业务主体特殊这一条标准，是一块非常复杂的业务，和现有其他业务的处理流程完全不同，通过微服务，将这个长流程业务拆出来，变成一个独立的服务。类似的业务主体还有一些，不再一一列举。

因为订单生成是两种渠道的方式，符合关键指标的不同渠道标准，这时拆分成两个服务，把通用的业务逻辑进行抽取，变成可以共用的组件，拆分成两个的好处是将这两种生成订单的方式进行解耦，变成两个独立的管道，当其中一个服务无法使用时，不影响另一个服务的正常使用，让客户有备选方案可以选择。其中一个上线时另一个不需要被捆绑着一起上线导致系统无法使用。

至此所有关键业务指标都已经识别出来了。

第五步：进行重构。

由于系统已经采用了前后端分离的方式，所以这里跳过此步骤，直接对后端进行重构。

重构步骤如下。

（1）因为业务量特别大，所以系统采用消息队列的方式进行处理，通过接骨法解除了代码之间的耦合。并提供了对外服务的接口，便于后面的上下游环节拉通。

（2）首先拆出所有的公共服务（校验服务、规则服务、开通服务、归档服务），然后再拆特殊业务主体服务（重点业务、长流程业务），最后拆不同渠道实现方式（接口服务和界面服务）。

（3）在创建订单环节，涉及所有的业务主体，影响大，所以创建订单的服务不能一下全部拆出来，要通过修路法，先建一个订单服务和原来的系统并行使用，同时分流出几个试点业务到订单服务里，当上下游调用没有问题，可以正常创建订单之后，再通过分批逃跑法，逐步地将其他业务的处理迁移到这个微服务上面，最终当所有业务都迁移成功之后，原来系统的订单处理逻辑就废弃了。

（4）在这些业务的拆分过程中，数据库拆分的工作也同时进行。

因为整个系统非常大，这里只列举了其中比较小的一个模块，麻雀虽小五脏俱全，非常具有代表性。

10.2 车队管理系统

10.2.1 背景

车辆信息管理系统，对所有车辆的信息进行管理，所属企业拥有全欧洲数量最大的车队。

10.2.2 技术架构

该系统主要涉及的技术有：Spring Boot+Spring Cloud 各组件；Yeoman JHipster Grunt Docker；Spring JPA；Spring security，等等。

系统架构图，如图 10-2 所示。通过 UI 登录，经过 ISAM 反向代理鉴权，就可以访问 Portal，前端再通过 Rest Services 调用后台的微服务，微服务可以调用第三方系统整合服务获得信息，也可以从数据库获取信息。

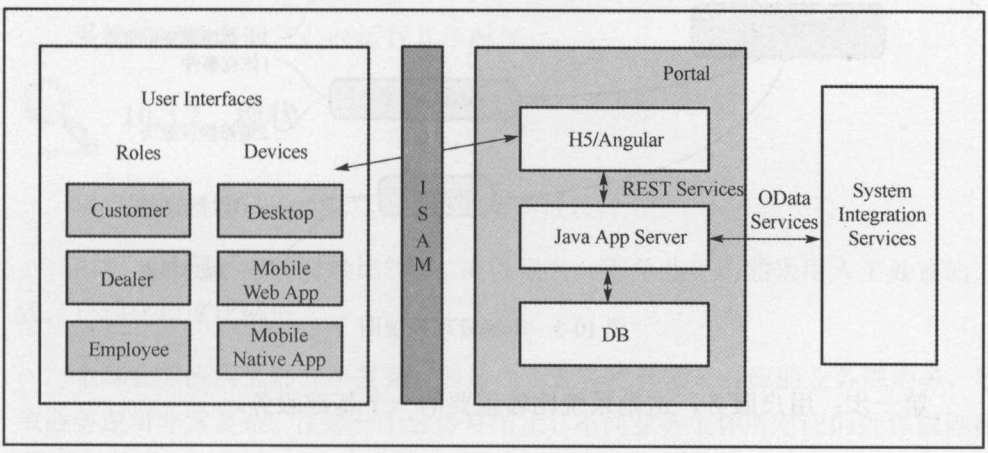


图 10-2　系统架构图



10.2.3 服务如何拆分

由于系统中不存在业务流程，只是基于车辆信息的管理系统，业务相对比较简单。

考虑的因素如下：

- 业务如何拆分；
- 服务之间如何调用；
- 粒度选择。

拆的时候就按照从无到有，从先到后的依赖关系来构建，这就要求先把基础服务创建出来。这样后面的服务创建之后就可以直接调用已经有的服务，而不需要再去新建轮子。

拆分过程如图 10-3 所示。

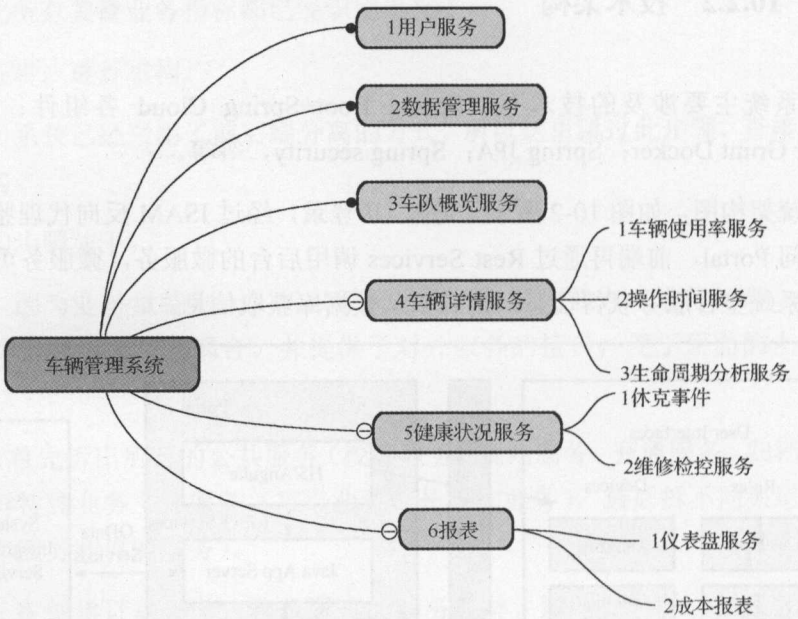


图 10-3　车辆管理系统图

第一步：用户服务，这是系统比较重要的一个基础服务。

第二步：创建数据管理服务，管理员可以录入信息，为系统提供基础数据。

第三步：创建一个车辆概览服务，用来显示车队概况。

第四步：从粗到细，创建针对车辆详情的服务，如车辆使用率服务、操作时间服务等。

第五步：将车辆的健康信息独立为一个服务，并细化成故障类事件服务及维修服务。

第六步：报表类服务，可以将车辆信息生成报表，有了车辆业务信息服务之后就可以被报表类服务调用生成报表。

这就是一个系统从无到有的整个业务拆分过程。

在粒度的选择上按照了互不影响原则。比如，健康服务和详情服务的拆分，本来健康服务是可以合并到详情服务的，但是因为此类信息从业务属性来说，侧重于故障，和基本信息又不是完全一样，所以单独拆分出来。

通过这个例子，我想大家已经知道拆分的过程如何进行。

10.3 财务管理系统



10.3.1 背景

这是一个核心的业务系统，用于财务的管理。

系统的架构老旧，已经运行几年时间。



10.3.2 现状

项目团队目前约 50 人，上线交付周期很长。

由于与外围系统的对接比较多，所以很大一部分业务仍然采用人工处置的方式，自动化程度比例低。

本身业务流程虽然并不复杂，但是由于业务主体多，对应的业务规则多，导致业务逻辑非常复杂。在每一个业务环节上，不同业务主体所对应的处理规则都是不一样的。

系统前端存在着特别多的业务逻辑代码，同时因为计算的逻辑非常复杂，都是通过采用存储过程进行处理的，数据库中存在几百个存储过程，个别存储过程的代码行数在 5000 行以上。

这样就让整个系统的前端和数据库端的逻辑特别多，形成了两头重的哑铃型结构。这种结构也是十分不健康的一种表现。

而处于中间的 Java 后台代码由于一部分逻辑是和存储过程相同的，就导致维护时要同时维护两套代码，为系统埋下了隐患。

以上这些情况导致开发起来特别烦琐，一个简单的需求需要修改的地方特别多，从前端到后端都要改动，牵一发动全身。同时由于所有的业务主体都在一起进行处理，这就导致耦合程度特别大，一个业务的修改稍有不慎可能影响所有业务，产生严重的 BUG。

而对于存储过程的维护来说，难度更是十分大，因为存储过程本身的可读性就差，只有对系统非常了解的人才能胜任，这就导致代码维护变成了对人的依赖。

综上所述，就是系统的现状。



10.3.3 原因调查

首先，系统采用的技术框架比较老，使用起来诸多不便。其次，就是架构缺少整体规划，导致后来整个架构的失控。再次，就是对于业务缺少规划的问题，没有将属于同一类别的业务主体通过泳道进行隔离。

最开始业务主体较少时没有考虑未来的扩展，后来随着主体增加也没有进行分类，还是按照以前的方式处理，只是增加判断逻辑，这样就导致后来业务主体特别多时，需要在前端和后端不断地增加判断逻辑进行处理，导致代码可读性特别差，整个代码乱成一团，维护难度也很大。



10.3.4 解决方案

针对之前架构缺少规划的问题，现在要重新进行规划，有针对性地解决现在存在的问题。

对于技术框架进行升级换代，对整个系统采用微服务的方式进行重构。

将整个架构从哑铃型变成纺锤形，将业务逻辑从前端和数据库收回到 Java 后端。

技术改造点如下。

（1）前端不再负责业务逻辑的判断和处理，只负责展示。

（2）对存储过程进行改造。

（3）通过服务化，与外围系统对接更方便，以达到提升自动化比例的目的。

业务改造点如下。

首先对现有业务流程进行梳理。

（1）将业务主体进行分类，通过泳道隔离，业务主体之间独立，互不影响。

（2）按照业务流转环节，将系统拆分成几个大的服务，如创建服务、审批服务、追踪服务等。

（3）其中规则逻辑符合业务复杂的标准，将业务规则进行整理，抽取成规则服务。

（4）计算模块消耗性能，影响大，同时又是核心功能，且不同业务主体都会调用计算的服务，它符合关键指标的标准，所以将计算模块拆分出来成为公共服务。计算模块的业务复杂性使我们不得不继续向下拆分，细分成正算和反算等多个服务，并将一些通用的校验逻辑也拆出来作为公共服务使用。

10.4 本章小结

通过上面三个案例的分析，我想读者已经可以了解业务如何拆分、如何使用微服务等内容。这里只是抛砖引玉，希望能给读者理论上的指导，具体实施时还要考虑自身系统的特点，做到灵活变通。



开发管理

微服务对于管理上的要求比传统应用还要高，如果管理做得不够好，可能会出现失控的风险，相当于练功走火入魔。如果项目管理做得不好，使用微服务，也是治标不治本。

由于项目管理的范围比较宽泛，所以这里聊聊更为精益的开发管理。

管理做得好，交付质量才会高；管理做得不好，交付质量一团糟。

11.1 管理原则

- 小团队易于管理，沟通成本低，交付质量容易控制；
- 尽量不要使用虚拟团队、远程协作；
- 部署服务时应该分批上，不要一次性上太多，容易定位问题。

11.2 日常管理

每天早上开一个站会，每人三点内容。

（1）昨天工作的进展；

（2）遇到什么问题；

（3）今天准备做什么。

组内多交流，采取众议的方式，讲给大家听听，有利于形成知识传播，共同进步。

11.3 代码质量管理

- 不断优化；
- 持续改进；
- 通过复用来降低代码出错的可能；
- 精益管理和持续交付。

有的项目中为了保证开发质量，由 SA 做概要设计，然后开发人员根据概要设计来开发。这里有个问题，就是开发人员需要等做好概设后才能开发，导致开发人员对 SA 的依赖，如果 SA 来不及设计，开发也会被耽误。会出现一大堆人在等一个人，这个时候就产生了依赖，应该采取依赖反转，让开发人员写概要设计由 SA 来审核。

11.4 工 作 方 式

累死人的不是工作，而是工作方式。

开发层面要学会规划和统筹，有些相同性质的工作，如都是增删改查的功能，就不要同时分给 N 个人来做，这样进度最慢 ，每个人都要花时间在编码上。转变一下思路，其实只要一个人做出来，其他人采用就可以了，这样效率才高。

开发时，如果程序员不了解系统，可能会重写轮子，或者写出逻辑很复杂的代码。这就要求开发之前，每个开发人员先将自己的思路说一下，如何实现，参考了哪些已有代码，不同之处在哪儿。

有很多项目不是没有计划，而是不按计划执行，所有的计划都延期，最后只能靠加班来还债。

11.5 BA 的职责

业务分析师（Business Analyst，BA）的职责：

- 需求明确；
- 业务的传播。

BA 简直是传道授业解惑的老师。每次新来了开发任务，都要做一次业务的授课。这种重复性的工作建议做成视频。

首先需求要明确。这个是入口，相当于吃饭，如果前面没吃好，后面消化起来很麻烦。让每个人知道要做什么和为什么做，这样才能更好地知道怎么做。

BA 确认需求，开需求评审，大家了解需求，BA 要合理地控制需求，随时了解排期完成情况，不能没有节制地接需求，要和 SA 一起评估工作量，需求接多了，超过开发的消化能力，就相当于长期处于暴饮暴食状态，质量必然下降。

通过 USER STORY 将需求进行文档化，开发人员看到 USER STORY 就可以很明确地知道需求的来龙去脉，USER STORY 里面如果包括了往来邮件的内容，这样就非常清楚了。

通过 JIRA 系统来管理需求，方便查找和维护需求。

11.6 SA 的职责

系统分析师（System Analyst，SA）的职责：

- 技术选型；
- 框架选择；
- 标准化工具；
- 代码质量管理；
- 审批概设；
- 提供工具，帮助开发人员快速上手。

涉及系统级别的需要重点关注。在需求出来时就要考虑可能对系统的影响，并且分等级，做好跟踪记录，确保部署时不出问题。

选择合适的技术框架可以极大简化开发难度，不要用一些冷门的技术，要用主流的技术，这样新人融入快，来之即战。

比较大的团队里因为人员多，可能做的事情比较独立，这时候工具类无法复用（除了时间等少数工具类外），这就要求架构在整体层面不断抽取出工具类，

并进行宣贯。这样有利于开发人员少走弯路，节省不必要的开发时间，而且用已有的工具更安全高效。

11.7 DEV 的工作原则

开发人员（Developer，DEV）应该遵循的工作原则：

- 先找轮子；
- 找不到时就开发可复用的轮子；
- 边开发边清理，像厨师做菜，做完菜，已经收拾干净；
- 边开发边写单元测试；
- 第三方库慎用，“请神容易送神难”，强耦合的集成，代码渗透太厉害；
- 尽可能增加代码，而不是修改现有代码。

业务功能一般照着做，有个参照物，最不容易出错。

开发的时候不仅要知道如何做，还应该知道为什么做，这就需要 USER STORY 文档的支持。

要经常对开发的内容进行讨论，避免双方理解不一致导致的问题。

11.8 本 章 小 结

微服务能否取得成功，达到应有的效果很大程度上在于管理，因为管理决定着最终交付的质量，微服务比传统应用更依赖于管理，需要有人来统筹、指挥，才能让微服务这支乐队演奏出时代最强音。

后　记

自从有了微博之后，曾经风光无限的博客就被自然而然地取代了。

现在很多以“微”为标签的新事物都格外受到追捧，微电影、微小说等，更不要说让人机不离手的微信，让很多人成了手机的重度依赖症患者。

而此时微服务受追捧的程度不亚于网红，未来它必将取代大多数传统架构成为主流架构，这是时代发展的必然结果。

选择使用微服务并非一蹴而就，这条路上可能布满了荆棘，如果真的选择这条路就要沉下心来，一步一步地向着既定目标前进。

好在这条路上的队伍越来越壮大，不会感到孤单，我们可以相互学习和交流，共同成长，必然可以到达成功的彼岸。

就让我们一起踏着荆棘上路，最终拾获鲜花。

微服务解惑篇

活字印刷微服务，顺应时代的召唤
得到容器神助攻，一身标签很拉风
单一职责海贼王，团队自治流程化
小小服务易扩展，实力碾压终成王
独立灵活速度快，全是组件易集成
解除耦合可复用，不足挑战应牢记
传统架构问题多，一剂良药微服务
诸多价值犹可见，快来定个小目标
业务敏捷高可用，持续交付高性能
业界大佬的选择，不微你就out了
拆分工作是关键，庖丁解牛是榜样
两个模型要清楚，金字塔图来帮忙
关键指标来识别，粒度适中边界清
如何规划学秦都，架构重构是难点
影响因素要规避，原则方法要掌握

技术实现篇

春云一站式服务，众多大牌来加持
生态丰富社区强，按需取用很轻量
注册中心Eureka，就像水浒梁山泊
智能网关Zuul，一夫当关可路由
客户端负载均衡，负责调度和选择
断路器Hystrix，变来变去魔术师
分布式配置中心，云端存储免重启
服务间调用Feign，假装自己在纽约
服务追踪Sleuth，私家侦探记录全
日志聚合Zipkin，一网打尽全链路
没有比较没伤害，Dubbo只是一小块

项目实战篇

三个案例不算少，多个场景都覆盖
如何拆分和使用，结合实际情况看
灵活使用最重要，切忌生搬和硬套
知行合一止于善，就算成功了一半

ISBN 978-7-121-31142-0

策划编辑：王二华
责任编辑：王二华
封面设计：周状伟

定价：65.00 元